An Object-Oriented Discrete-Event Simulation System for Hierarchical Parallel Simulations

THESIS

Kenneth W. Stauffer
Capt, USAF

AFIT/GCE/ENG/96D-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/96D-02

An Object-Oriented Discrete-Event Simulation System for Hierarchical Parallel Simulations

THESIS
Kenneth W. Stauffer
Capt, USAF

AFIT/GCE/ENG/96D-02

AFIT/GCE/ENG/96D-02

An Object-Oriented Discrete-Event Simulation System for Hierarchical Parallel Simulations

THESIS

Presented to the Faculty of the
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Kenneth W. Stauffer, B.S.E.E.
Capt, USAF

December 17, 1996

Acknowledgements

First I would like to thank Dr. Hartrum. His insights into parallel simulation and modeling helped me to greatly expand my knowledge in this arena. Secondly I would like to thank my committee members Lt Col Wailes and Maj Banks for their inputs.

I want to thank my classmates and friends, Capt Glenn Jacquot and Capt Brian Garcia. They were always willing to make a "run for the border" when a break for dinner was in order.

I want to thank the former AFIT graduates who developed and adapted BattleSim. From this group, I would especially like to thank Capt Jim Hiller for the hours he spent helping me to understand the C programming language while I was debugging my new architecture and the insight he provided about the way BattleSim was constructed.

I would like to thank my parents for making me realize how important education is and for instilling an attitude of *Never Quit* into me.

Last, but not least, I want to thank my girlfriend Kelly whose love and understanding during the last few months of this research helped me to see the light at the end of the tunnel.

Kenneth W. Stauffer

Table of Contents

List of Figures

List of Tables

Abstract

The purpose of this research is to design and implement an object-oriented discrete-event simulation system which supports hierarchically constructed players in a parallel or distributed environment. This system design considers modularity and portability so additional modules may be implemented to experiment with new algorithms for both partitoning and synchronization.

A simulation system which meets these requirements was partially implemented on an eight-node Intel Hypercube in C. A desired goal was to maintain the functionality of the existing BattleSim application. Test cases used measure the performance and correct operation of the new simulation architecture using a BattleSim subclass. Test results prove correct operation of the new architecture, but show a significant slow down in the parallel operation of this system.

An Object-Oriented Discrete-Event Simulation System

for Hierarchical Parallel Simulations

I. Introduction

1.1 Background

Due to the cost of running tests with aircraft and weapons systems and the ever increasing capabilities of computer systems, there has been a push to model these aircraft and weapons systems on computers in the form of simulation. Once established, these computer simulations cost much less than the operational costs of real aircraft and weapons systems. One drawback to computer simulation is the time required to run a simulation which accurately models a real system. This has led to performing computer simulations in a parallel or a distributed environment in order to reduce run-time of the simulation by utilizing multiple computer processors.

In order to produce a realistic model which will produce realistic simulation output, each major system of an aircraft or weapon system must be represented in a simulation. This usually requires complex code which is not easily reusable between models (i.e. F-16 vs F-14 require an equal amount of work to model). This reuse problem is solvable by applying the object oriented paradigm to design generic systems whose variables can be loaded at runtime from configuration files. Each model is built through aggregation and inheritance of these systems.

At the Air Force Institute of Technology, a military simulation called BattleSim has been developed in order to model multiple players in a spatially partitioned battlefield (3, 10, 12, 17, 20). This system currently uses a conservative approach to processor synchronization and uses only

simple non-hierarchical players. Masshardt developed an independent hierarchical based player (a tank) in 1995 (12). In order to increase flexibility and enhance reusability, it is desirable to incorporate this complex hierarchical player structure into BattleSim. This complex player will be required to interact with existing simple players with complete transparency to the user. In other words, the user of BattleSim should not be concerned if a player is simple or complex. Since a complex player may take longer to simulate than a simple player, a mechanism must be designed to optimize the simulation of a complex player. This optimization must be transparent to the user. Lastly, in order to minimize modeling of military systems within BattleSim, a mechanism/interface to allow other Distributed Interactive Simulations [DIS] to participate with BattleSim players through the DIS architecture must be developed.

1.2 Problem

The large scale implementation of simulations throughout many organizations has introduced many methods of simulation. It is desirable to develop a common architecture which many simulations could adopt to increase reuse among various simulation environments. The creation of the DIS architecture allows simulations from different locations to participate with each other, thus minimizing work within organizations. To expand the functionality of BattleSim, it is necessary to design an interface to allow interaction with DIS players. To make BattleSim more appealing to other organizations, it is also necessary to increase the complexity of the current players which will allow them to be more realistic.

Problem Statement

- Design a simulation architecture which supports multiple hierarchical players on a parallel computer.
- Design an interface to allow DIS players to interact with the players of the new simulation system.

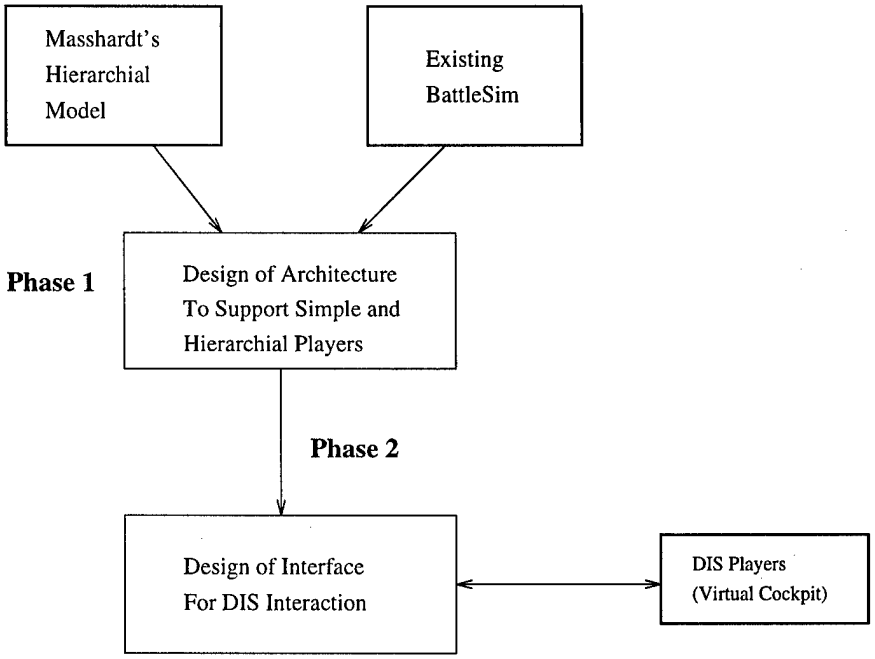

Figure 1. Big Picture of Problem

Figure 1 shows a procedural model of how the problem can be divided into two different phases.

1.3 Initial Assessment of Past Efforts

Due to the lack of an easily modifiable, unclassified military simulation, Rizza developed the original version of BattleSim. Although parallel issues were considered while building BattleSim, Rizza did not parallelize BattleSim during his thesis work. The original BattleSim ran on a single

serial processor using simple non-hierarchical players. BattleSim was written in C and is capable of running multiple scenarios with multiple simple players limited only by the memory available on the computer on which it is executing (17).

In 1992 Bergman parallelized Rizza's BattleSim code. He used spatial partitioning with a conservative approach to processor synchronization. He also used a hierarchical, object-based approach while building the structure of BattleSim (3).

In 1993 Trachsel investigated an object oriented approach to parallel simulation using BattleSim. His research primarily dealt with the OO representation of the simulation system and not with the OO representation of a complex player (20). Trachsel created object-based modules of the BattleSim code as much as possible but still left a lot of the objects highly dependent upon many other objects. This causes a lot of dependacy between the simulation layer and the application layer, making the original BattleSim code highly unusable for other simulation projects due to the nature of BattleSim specific code which is embedded in the simulation layer.

Figure 2 represents Trachsel's object based architecture of BattleSim. The actual communication and dependency model of BattleSim was analyzed to verify the accuracy of this model. Trachsel's model, based on Garlan and Shaw's object oriented organizational model, does not account for key features of the software architecture. These features include structural issues for a global control structure, protocols for communication, synchronization and data access (8). These components of BattleSim were also analyzed during this thesis effort.

In 1994 Hiller developed analytic performance models for BattleSim. His work kept the conservative synchronization approach already developed in the parallel implementation of BattleSim. He also developed a scenario generator with which to generate test cases (10).

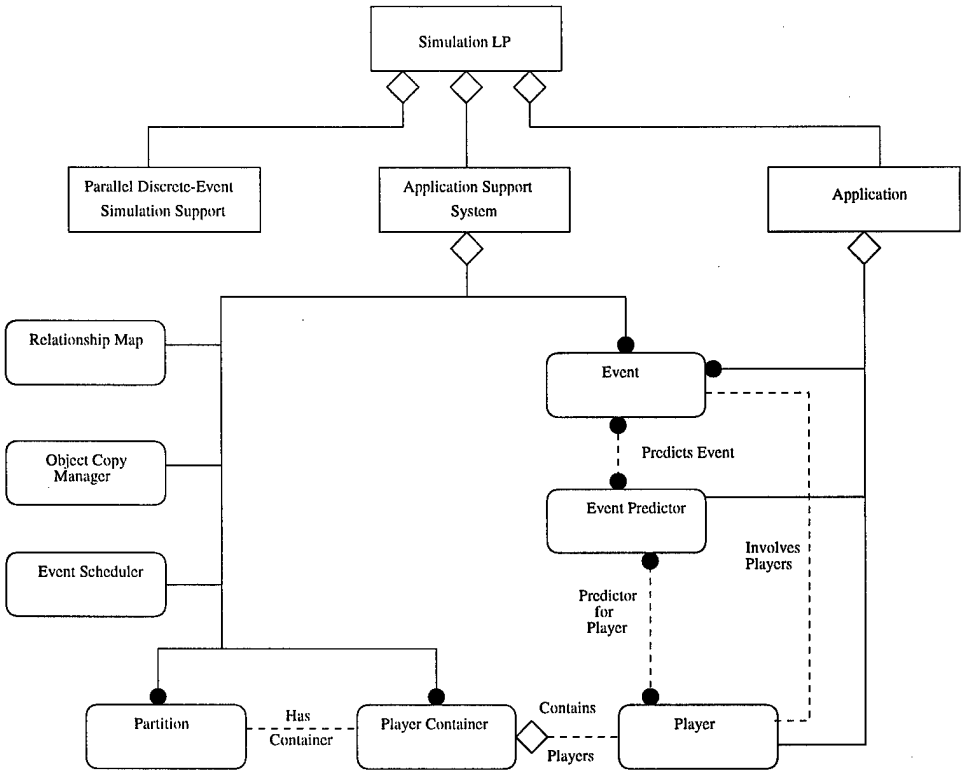

Figure 2. Object Based Architecture of BattleSim (20)

In 1995 Masshardt developed a complex hierarchical model of an Army tank. His goals were to study the object partitioning of this complex object oriented model to determine how to best partition the complex model in order to reduce run-time. In his efforts he developed his own simulation environment (12). This environment (which was good for his tank) does not allow for additional scenarios to easily be added to his simulation architecture. Reconfiguration of the tank simulation requires code changes instead of reading the scenario in from a file. His model also only allows for a single tank to be modeled instead of simulating several tanks in a scenario. This simulation architecture would be very difficult to transform into a general purpose simulator, but his hierarchical model is a commendable start for future hierarchical models. Masshardt's tank is further discussed in Section 2.7.

1.4 Scope

Although a hierarchical model may be more complex than the simple players currently existing in BattleSim, the goal of this research is to show how this architecture can work to maintain or increase simulation performance and to add flexibility, and to allow multiple domains to be modeled. It was not the goal of this research to accurately model, to full detail, a military system, but only to model computational loads associated with real systems.

1.5 Approach

There are three significant goals of this research.

1.5.1 Phase 1. Integrate a complex hierarchical player into BattleSim.

- Perform a literature review to examine current technologies in object-oriented simulation and hierarchical modeling.
- Study the BattleSim code and design a simulation architecture which will support both simple and hierarchical based players. which will allow multiple domains to be simulated.
- Measure performance of the new simulation system with combinations of simple and complex players.
- Investigate implementation order of proposed Phase 2A and Phase 2B.

1.5.2 Phase 2. Develop an interface layer to communicate with a DIS manager.

- Interface with AFIT's DIS systems in the graphics lab.
- Allow other DIS players to see and react to the new simulation players.
- Allow the new simulation players to interact with DIS players.

1.6 Outline of Thesis

This chapter includes the problem statement and approach taken to solve the three problems stated. Chapter II provides a clearer picture of some of the problems faced in the form of a literature review. Chapter III is an initial design of the proposed solution including discussion of potential problems. Chapter IV discusses the building of the new simulation architecture including a discussion on reuse of the existing BattleSim code. Chapter V includes test cases which verify the functionality of the new architecture in comparison to the original BattleSim architecture. Chapter VI discusses the conclusions of the work done and areas of interest which require more research to

improve the existing architecture and to add more functionality and flexibility to this simulation architecture.

II. Literature Review

2.1 Introduction

As discussed by Maj General Joseph J. Redden in the keynote address to the 1995 Winter Simulation Conference, "computer simulation and modeling can be used as a decision support tool to determine how a battle force should be constituted and how it should be deployed" (16). Due to this driving force within the military, the growing demands of simulation must utilize new technology to meet the needs of military customers, producing faster, more accurate, and useful military simulations which "represent the increased complexity of modern combat" (14). In order to better understand these requirements, this chapter focuses on simulation, parallel issues and software architectures which influence military simulation.

2.2 Simulation

A simulation is defined as "the imitation of the operation of a real world process or system over time" (1). Simulation can be classified in two different ways: *continuous* and *discrete*. Continuous simulation involves observing a model in real-time; this does not conform well in a computer simulation due to the fact that time within a computer must be incremented in steps, thus in a discrete or step by step implementation. Continuous simulation is best applied to non-computer modeled simulations. Discrete simulation involves updating actions and positions of simulated components based on some constant τ. Discrete simulation conforms well to computer simulations but has a distinct drawback. The simulation incrementation time τ is held constant, causing the simulation time to be pre-determined for a finite length simulation. During these incrementations of

τ, there is no guarantee that a major event will take place in the simulation for each incrementation of τ.

To overcome this constant simulation incrementation time a method called Discrete Event Simulation can be implemented. In Discrete Event Simulation the simulation clock is not updated based on some value τ, but is updated based on significant events within a simulation. These events could include an aircraft reaching a certain route point, a missile being launched or an aircraft running out of fuel, etc. Basing the simulation on events vs. a constant τ, the simulation time may be greatly reduced.

Simulations currently are limited because, in order to receive an accurate answer to a problem, the item being simulated must be accurately mathematically modeled. Without an accurate mathematical model, simulations may not give a result which is accurate in the real world system that is modeled. Another limitation to simulation is the response time for the appropriate problem. Some organizations within the military use simulations to make wartime decisions (13, 14, 15). If a simulation is run to determine the best egress and digress paths of a particular target for a mission, the simulation must be completed before the mission is executed. For this reason, the technology of parallelizing simulations must be studied.

2.3 Parallel Issues

Parallel Discrete Event Simulation is a Discrete Event Simulation in which the simulation is executed on more than one computer processor. With the introduction of multiple processors, there are two distinct problems which are to be faced: *partitioning* and *synchronization.*

2.3.1 Partitioning Schemes. Partitioning of a simulation confronts the key issue of "How does one distribute the simulation between multiple processors?". There are two basic approaches to this problem: *spatial partitioning* and *object partitioning.*

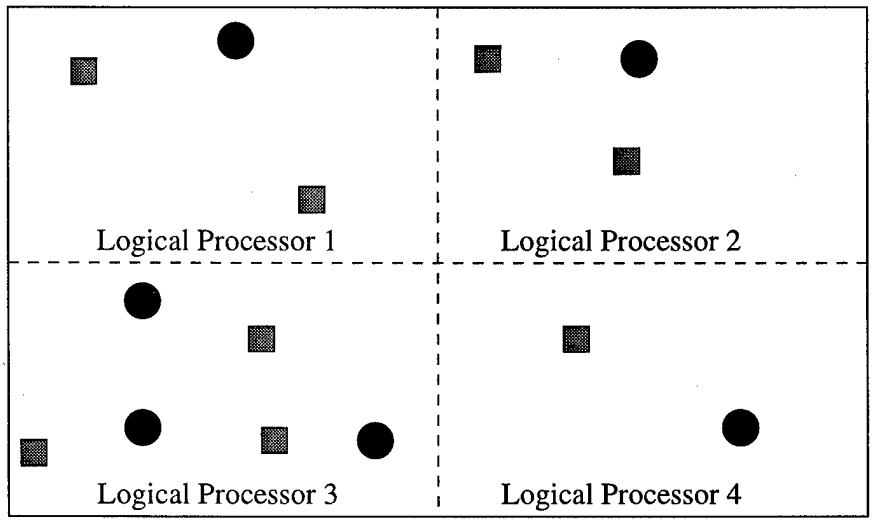

Figure 3. Spatial Partitioning of a Battlefield

2.3.1.1 Spatial Partitioning. Spatial partitioning requires that bounds be placed on the playing field of a simulation. A classic example of this is the simulation of a pool table. The table surface would be the boundary of the simulation. To gain benefit from parallelization, the pool table surface is divided into partitions based on the number of computer processors dedicated to the simulation. The pool balls in this example are simulated. Each pool ball is assigned to a computer processor based on the location of the pool ball in the simulation. Figure 3 shows how spatial partioning works with respect to a battlefield simulation utilizing four processors. Spatial partioning has the advantage of reduced interprocessor communication since processors only have to communicate when a pool ball crosses a boundary and is getting assigned to a different processor. The disadvantage to spatial partitioning is the potential for unbalanced processor loads. This can

be caused by too many pool balls in one area of the table, causing one processor to work harder than the others. Figure 3 shows a heavy load on processor 3.

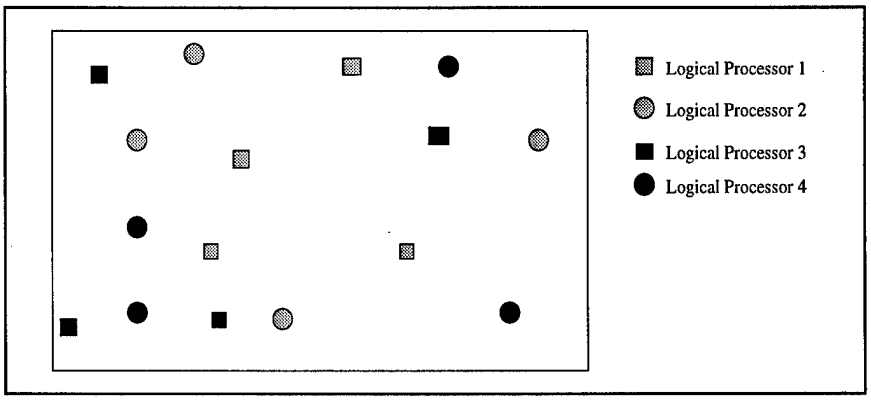

Figure 4. Object Partitioning of a Battlefield

2.3.1.2 Object Partitioning. Object partitioning, unlike spatial partioning, does not require bounds to be placed on the playing area. This is because object partitioning distributes the workload based on the number of objects or players in a simulation. Using the pool ball example, each ball is assigned to a processor. In a four processor simulation each processor is assigned four balls. Since the complexity of each object is similar, each processor has an equal amount of work. Equal load balancing is the main advantage to object partitoning. The disadvantage of object partitioning is the overhead of communications between processors to keep track of positions of the balls in the simulation playing field. Figure 4 shows an object partioned simulation using four processors.

2.3.2 Synchronization Schemes . Synchronization between processors is required to maintain the causality of events within a system. Causality is the proper time ordering of events (19). Two methods to ensure causality are *conservative* and *optimistic* synchronization.

2.3.2.1 Conservative Synchronization. According to Fujimoto (7), conservative synchronization was the first type of synchronization algorithm to be developed. Conservative synchronization ensures causality of events within a system. However Fujimoto also points out that conservative synchronization is prone to deadlock. This places a requirement upon conservative algorithm developers to ensure two mechanisms: causality between events and deadlock avoidance. Deadlock avoidance requires that the "maximum resource requirement of a process be known at every point during its execution" (19). Resources are only granted to processes if the process can be guaranteed to finish with the resources available.

One of the most popular conservative algorithms was developed by Chandy and Misra (4). Their method works on a concept of the "null message" routine. After every event execution, a timestamped message is sent to all other processors. This null message allows other processors to develop a timestamp baseline to determine if their next event can be processed. This "null message" system prevents deadlock. The Chandy and Misra algorithm, however, is a static algorithm. This means that the number of LPs in the simulation must remain constant and be known prior to the beginning of the simulation. Causality is enforced in this algorithm by requiring each message stream coming from other LPs to carry events in timestamp order. The reception of these messages are stored in a FIFO queue to ensure they are processed by the receiving LP in time-stamp order (4).

2.3.2.2 Optimistic Synchronization. The main difference between optimistic synchronization and conservative synchronization is that optimistic synchronization allows causality of events to be "disrupted" or to be received out of temporal order. but has a mechanism to "roll back" the simulation in order to recover. The main advantage optimistic synchronization has over

conservative synchronization is that optimistic synchronization allows events to be executed concurrently on different processors, thus exploiting parallelism to a greater degree. It does, however, add overhead to ensure causality which may cause performance degradations. One of the best known optimistic algorithms is known as *TimeWarp*, developed by Jefferson (11).

2.4 Simulation Architecture

In order to increase modularity and reuse of a basic simulation system and to allow multiple types of simulations, an architecture or building plan must be designed. Garlan and Shaw describe the components of software architecture as being: structural issues which include organization and control structure, communication protocols between modules, functionality of design units, and distribution of modules (8). Three basic architectures which are commonly used include: *pipe and filter*, *object oriented* and a *layered approach*(8).

2.4.1 Pipe and Filter. In the pipe and filter architecture, components are linked together each having a defined set of inputs and a defined set of outputs. The pipe and filter architecture has three advantages: understandability, reuse and maintenance. Disadvantages of the pipe and filter architecture include a batch processing approach where one module must wait on the output from another module, thus causing a delay. They may be hampered by maintaining several different but related data streams. and lastly they may add work in the form of parsing and unparsing data in each module or filter (8).

2.4.2 Object Oriented. In this representation, objects represent abstract data types which communicate with each other through function or procedure calls. Objects are often privatized to maintain data integrity of the object. Advantages of an object oriented approach include reuse,

maintainability and understandability (8). As long as the interface to an object does not change, implementation of the object may be changed without affecting clients of the object. The object oriented approach also allows a problem to broken into smaller pieces. The primary disadvantage to an object oriented architecture is that objects must be aware of other objects with which they interact (8).

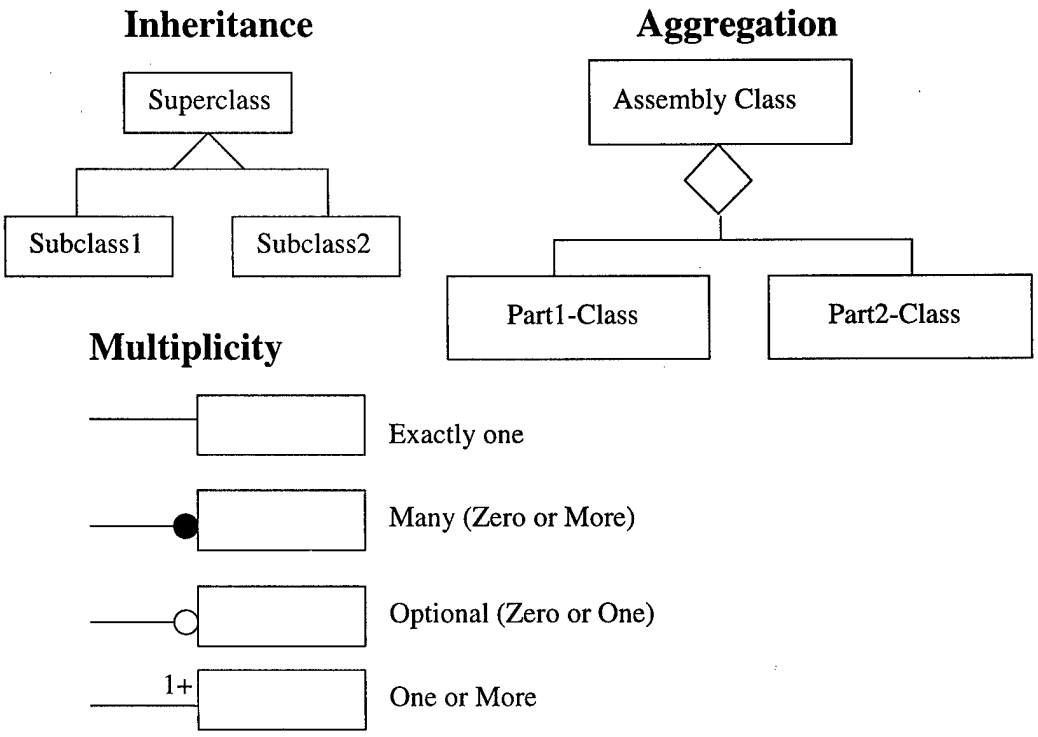

Figure 5. Rumbaugh Diagram Examples

Rumbaugh defines object oriented modeling of a system as an organization of software using a "collection of discrete objects that incorporate both data structures and behavior" (18). Characteristics of object oriented modeling include: *identity, classification, polymorphism, inheritance,* and *aggregation.* Identity is the process of quantifying data into distinguishable entities which are objects. Classification is the identification of objects with identical data structures and behavior. Polymorphism is the concept of an operation which seems the same by name but operates differ-

ently between classes. Rumbaugh compares the move operation between a window class and a chess piece class (18). Inheritance is the sharing of data structures and behavior based on a hierarchical relationship. In this relationship, the child structure has the same operations as the parent to which the child belongs. Aggregation is the formation of a single object or class by composing two or more classes. The concepts of class, inheritance and aggregation are shown in Figure 5 as represented by Rumbaugh.

2.4.3 Layered approach. A layered approach is typically organized hierarchically with each layer providing services to layers both above and below it. A layered system provides the advantages of easy enhancement, reuse and the ability to break large problems into a smaller level of abstraction. Disadvantages include the fact that not all systems can adhere to a layered approach since they might need information from a layer more than one level away. Levels of abstraction may also be more difficult to define for the same reasons (8).

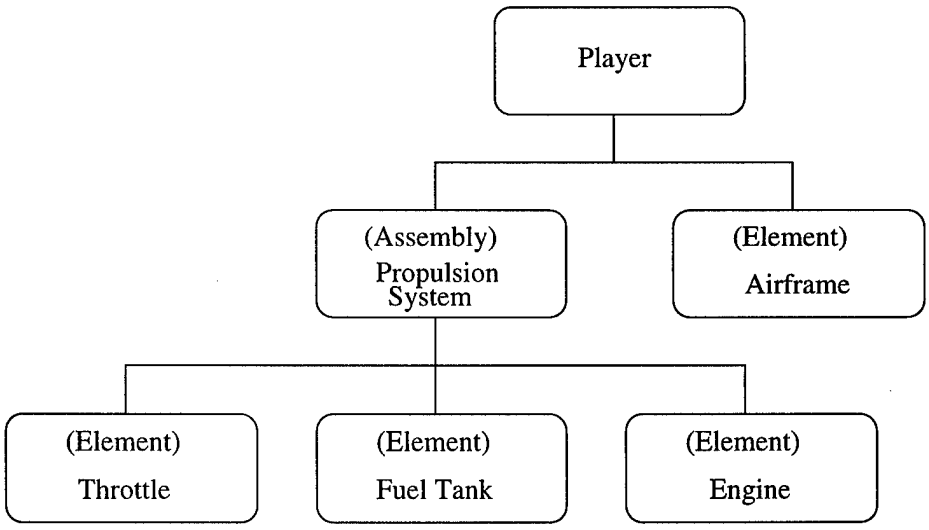

Figure 6. JMASS Player Representation

2.4.4 JMASS. The JMASS (Joint Modeling and Simulation System) is based on the SSM (Software Structural Model). This allows an object to be represented as a hierarchal partitioned model of objects. The SSM is based on the OCU (Object Communication Update) model which defines one call to each model. This call is known as "update"; all data required for updating the model is passed to the model during this call. The update call in turn calls all functions or procedures within an object in order to set it to a proper state. The SSM model deviates from the OCU model by allowing four types of events. These events include: RF communications, Detonate, Spatial and Null. Simulation control within the SSM includes a package which will handle event generation, event handling, environment update and player update. The environment is treated as a player in the SSM model. It tracks the state and location of all other players in the simulation (21).

JMASS modeling includes three layers of abstraction. These are *Player*, *Assembly* and *Element.* The composition of an example JMASS player is shown in Figure 6. An element is the lowest level model component. In a highly abstracted model an element may be a switch or a tire. An assembly is a collection of more than one element. A propulsion system assembly may be comprised of the following elements: *engine*, *throttle* and *fuel tank.* Lastly a player is an assembly that may be comprised of many other assemblies or elements. The player has a direct link to the simulation system and interacts with the system for all of its sub-assemblies and elements (21).

2.4.5 DEVS. DEVS, also known as Discrete Event System Specification, is a formalized structure for developing object oriented simulations in a hierarchical manner (22). The DEVS Scheme is specifically designed for discrete-event model construction and simulation. DEVS uses two approaches to models, *atomic models* and *coupled models* (23). Atomic models in DEVS

follow the set-theoretic formalism for discrete event models which was developed by Ziegler. There are four associated calls with the set-theoretic formalism: *internal transition function, external transition function, output function* and *time advance function.* Coupled models are broken into three categories, *class coupled models, class broadcast models* and *class digraph models.* Class coupled models are defined in relationship to children of a class. Four basic calls are used for transfer of information between parents and children, *get-children, get-influencees* (determines to which children specific output is sent), *get receivers* (determines receivers of external events directed to the parent) and *translate* (provides communication port translation) (23). Class broadcast-models allow all components to talk to internal components and the outside world. No limitation is made to whom the message is sent. Class digraph-models are a hybrid of coupled models and broadcast models in which communications to other modules can be controlled and limited. (23)

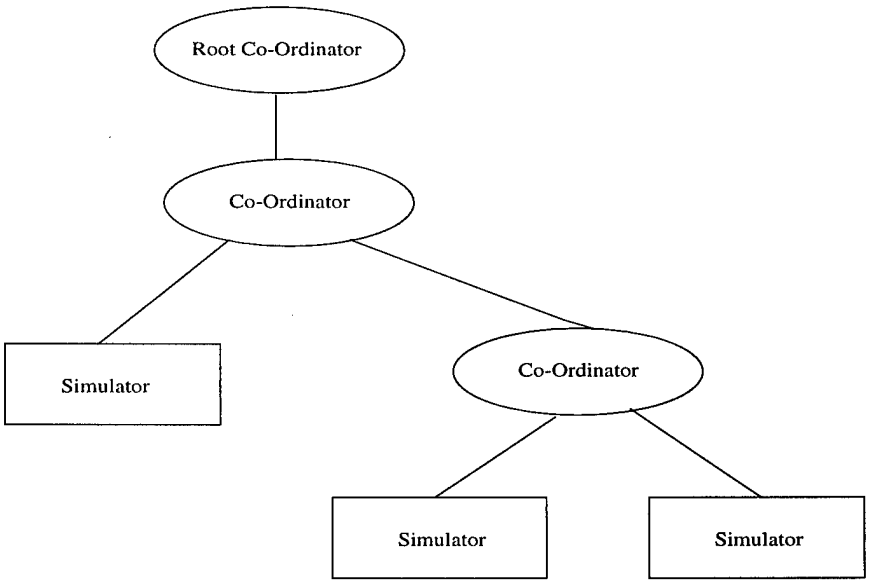

Figure 7. DEVS Simulation Model

Simulation within DEVS requires each object to maintain its own next event queue. This is done by setting up a *root-coordinator* for the top level parent and setting up a *co-ordinator* in

each object in the tree/graph of a hierarchical model and a *simulator* for the lowest level object (23). This adds a layered approach to the concept of an object-oriented software architecture. Figure 7 shows a model of how the DEVS simulation structure may be used with regard to the *root co-ordinator*, *co-ordinator*, and *simulator* (23).

2.4.6 PASE. The Parallel Ada Simulation Environment (PASE) was developed by Belford in 1993 at AFIT. The PASE model was implemented in Classic ada and followed the JMASS architecture. Although hierarchical models of players in the simulation are mentioned, hierarchical players were not implemented or designed (2). Belford also does not describe the possible interaction with a DIS player nor does he describe the use of different partitioning schemes. Although the model proposed by Belford does not use any of the existing architecture or support functions described by the original BattleSim, it appears that the existing BattleSim could easily be migrated into Belford's model.

2.5 TCHSIM

TCHSIM is a general purpose discrete event simulation environment which allows the experimentation of several application models without recreating the basic structures every time (9). TCHSIM was also written to interface with the UVA SPECTRUM protocol in order to hide parallelism at a low level, thus being transparent to the user. Three basic components which are platform independent are the: *clock, next event queue (neq),* and *event* (9). The clock provides four basic operations:

- init_time
- set_time

- get_time
- advance_time

The next event queue maintains events in time order allowing for events to be added and removed from the queue. The Next event queue provides eight basic operations:

- show_neq
- add_event
- count_event
- neq_time
- get_event
- peek_event
- simultaneous
- max_neq

The event object provides calls for interactions of events of specific types at a specific time for one to three players in a simulation.

2.6 BattleSim Analysis

Figure 8 shows the current communication dependencies between the object modules in the original version of BattleSim. Even though any communication diagram can be drawn into a spiderweb of lines, the manual process used to construct this diagram was meant to keep the drawing as simple as possible. As shown in Figure 8 there are many dependencies between modules. As part of the analysis of the existing BattleSim architecture, communications were analyzed to

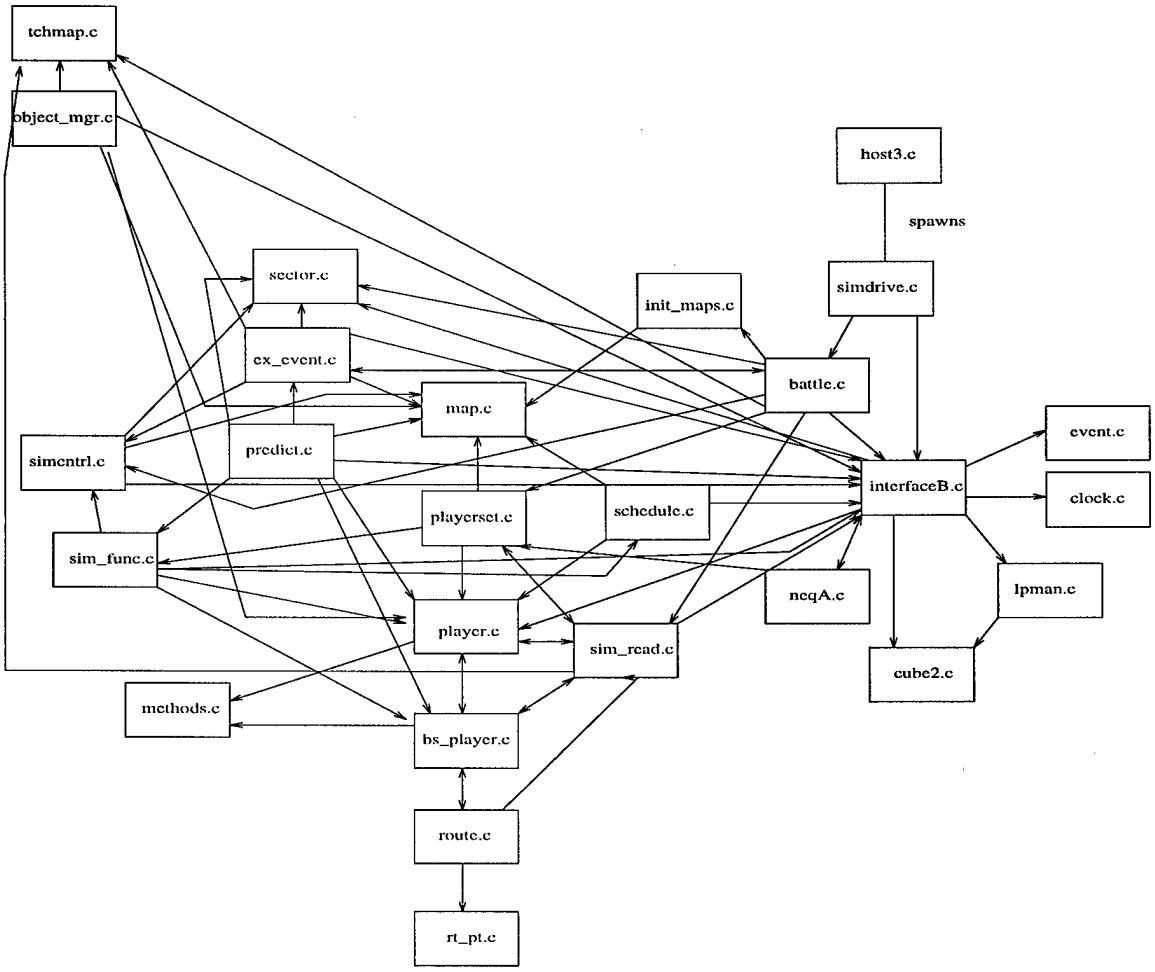

Figure 8. Original BattleSim Architecture

reduce dependecies between modules but also to maintain the functionality of the original BattleSim project. Section 4.2 discusses in detail the mapping of the current BattleSim modules into the new simulation architecture.

2.7 Masshardt

In 1995 Masshardt developed a simulation for an object-oriented model of a tank. This object-oriented model is decomposed in the simulation as an object partitioned discrete-event simulation using a tree of aggregate event queues. This method of using aggregate event queues is very similar to Ziegler's DEVS approach. However, communication is limited to parent to child communication which was chosen for ease of implementation and initialization. This structure is similar to the DEVS class-coupled approach discussed in Section 2.4.5. Also, similar to the DEVS approach, each object within the simulation has a time-ordered NEQ. Each queue contains events for the object which the event prediction generates after receiving an *update* call and the next event from its child, which is similar to the JMASS approach. Masshardt describes the simulation progressing as "a wave travelling down the hierarchy, bouncing back up and down any number of times before returning ... to the top simulation object" (12). By analysis of his description, the algorithm used to implement this traversal of the tree seems very similar to the *depth-first search* algorithm as defined by Cormen (6).

To update the simulation in a forward time direction, events must be processed in a causal order which does not exceed the current simulation time. For example: if the simulation time is (τ) then all events with time (τ-x $\leq \tau$) must be processed. All objects must have a method to process their events and should have an event handler to determine unknown events which are from their

children. This event handler for unknown events should only be able to determine which child the event is from and should allow the child to process the event (12).

Masshardt uses a form of conservative synchronization for his tank for three specific reasons. These reasons include: AFIT's research thrust is conservative synchronization, simulation state is always correct, and storage space for past events is minimized. His conservative algorithm seems similar to Huang's termination detection algorithm as described by Singhal (19). Each processor contains an object partition of different aggregates of the tank. Each processor will only process events for that processor once the parent of that partition reports to the simulator (12). This ensures that the parent knows the event status of all of its children and grandchildren and has the minimum event time possible. This characteristic is similar to Huang's termination detection algorithm.

2.8 Distributed Interactive Simulation (DIS)

Distributed Interactive Simulation (DIS) is a standard infrastructure which allows the creation of a large virtual simulation environment in which many persons interact. DIS allows persons with different simulated objects to interact together through a computer network thus allowing people at remote sights to participate in a single simulation. One concept of DIS is to allow an Army organization that simulates tanks to have its tanks participate with an F-16 model created by an Air Force organization. Capabilities of the DIS standards include definitions for:

- Identification of data items
- A common representation of these data items
- Formatted Messages called Protocol Data Units (PDUs)

- When PDUs are transmitted
- Processing of PDUs
- Key algorithms (e.g. dead reckoning) for all participants

Historically DIS has been associated with continuous, real-time, human controlled simulations. The DIS Steering committee also plans on having DIS interface with more automated simulations such as the Air Force simulation *Air Warfare Simulation (AWSIM)* (5).

2.9 Conclusion

This chapter primarily focused on recent research performed in the parallel discrete-event simulation field with respect to partioning and synchronization algorithms. Several software architectures were also presented in order to provide a common framework for building simulation systems. In order for parallel simulation to advance with growing technologies, one must use these methods as a baseline and develop hybrid models to gain maximum benefit from each type of algorithm.

III. Design Considerations

3.1 Introduction

The design of a reusable object-oriented simulation system that is capable of representing multiple types of simulations must have a well defined architecture. This chapter covers topics of interest in defining an object-oriented simulation that is capable of using both spatial and object partitioning schemes. Object representation and interaction with the basic simulation architecture are discussed and possible alternatives are described. The overall architecture uses components of JMASS, TCHSIM, Ziegler's DEVS architecture and Belford's PASE model as a baseline.

3.2 Layered Approach

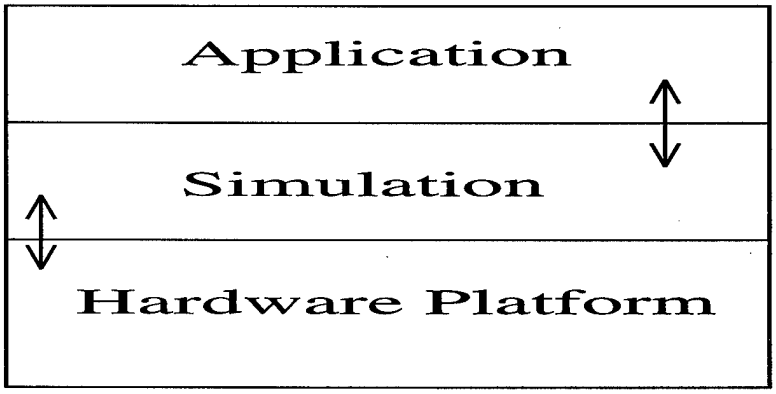

Figure 9. Layered Approach

Figure 9 shows a three level approach to layering the simulation system. The application layer should be written so that no dependency between the application and hardware exists. The simulation layer will interface with the application and the specific hardware platform. Removing dependences from the application layer of all hardware specific calls will allow applications to run on any hardware platform which has a compiler in which the application was written. The simulation

layer provides a generic interface to both the hardware layer and the application so that many different types of applications can be used with the simulation system. Using Generic calls to the hardware layer will also allow many different hardware platforms to be used with this simulation model without modifying the internal structure of the simulation layer.

3.3 Simulation Model

The generic simulation model represented in Figure 10 is able to handle any type of parallel simulation application without regard to the programming of the application. Only a basic set of functions are required to be present within the application itself to communicate with the simulation system. This adds transparency to the programmer of future applications with regard to parallel issues. The following sections describe the proposed interaction of modules within the simulation system and discuss options available to implement correct operation of the whole system.

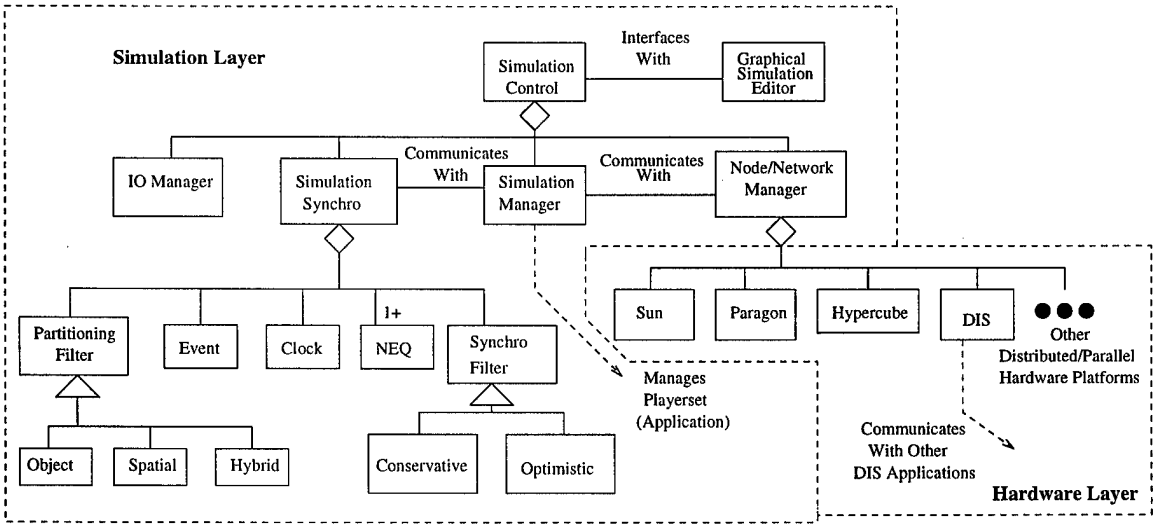

Figure 10. Simulation Layer

3.3.1 Simulation Control. The Simulation Control Module is the main interface to initializing the distributed/parallel simulation environment. This module is responsible for calling the

simulation manager once the environment is initialized and ready for the simulation and application to be instantiated onto the distributed/parallel environment.

3.3.2 Graphical Simulation Editor. The graphical simulation editor is a module which expands the ease of use to the simulation user. This module will to allow a user to use a point and click interface in order to design the simulation and to start or stop the simulation. This module is not part of this research effort. This simulation editor could be linked to a Knowledge Based Software Engineering (KBSE) module which will allow simulations to be created from the domain the KBSE represents.

3.3.3 IO Manager. This section of the code handles all file input and output. Upon initialization of the simulation all input data files will be opened and their address will be passed by reference to the application so that the application data can be read. A log indicating events, simulation time and details of the application will be opened. This file will collect pertinent simulation data in order to record the actions of the simulation.

3.3.4 Simulation Synchronization. This is a transparent interface between the simulation manager and the aggregate components that compose the simulation synchronization class. This module interfaces with the following components: *partitioning filter, clock, event, NEQ,* and *synchronization filter.*

3.3.5 Clock. This is a basic clock used to increment the time on each LP or processor which will keep the simulation moving in a forward direction. The clock will be accessed by the *simulation synchronization* and *next event queue.* It will also be used by the *simulation control*

module in order to pass the correct simulation time to the application. The design of this basic clock comes directly from TCHSIM (9) and is discussed in Section 2.5.

3.3.6 Next Event Queue (NEQ). The design of the next event queue is adapted directly from the TCHSIM project. However, this NEQ is only capable of having one copy per processor in a distributed system or node in a parallel machine. In order to adapt to either a JMASS approach, which maintains one NEQ in the *environment* object for all players on a processor or node, plus an additional NEQ for each player, or a DEVS approach which maintains a NEQ for each hierarchical component, the TCHSIM design must be modified to have an operation which *instantiates* a new NEQ in order to allow multiple NEQs for each object in the simulation. The operation between these multiple NEQs is discussed in Section 3.6. The operation of the NEQ is discussed in Section 2.5.

3.3.7 Event. This module is a generic representation of events whose event types can be inserted in a generic fashion to allow multiple event types from the simulation domain. This event module was adapted from TCHSIM and is discussed in Section 2.5. This model will allow one to three entities to interact with each other. For the purposes of this research this number of interactions is sufficient.

3.3.8 Synchronization Filter. This section of code determines the synchronization algorithm used for the parallel implementation of the simulation system. This filter will require communication with the *clock* and *NEQ* to determine if the simulation is in a safe state and will give authority for the simulation to continue. This is assuming a conservative synchronization approach. Further expansion of this code will allow optimistic synchronization to include a rollback

feature and will also allow hybrids of conservative and optimistic synchronization. At this time, it is only desired to implement a working conservative synchronization algorithm.

3.3.9 Partitioning Filter. The Partitioning filter maintains the proper algorithms for both spatial and object partitioning and will also allow hybrid models for partitioning. The operations within the *object*, *spatial*, and *hybrid* modules will be called through the Partitioning filter. This module is called directly by the simulation synchronization module in cases of event prediction from the application.

3.3.10 Simulation Manager. The *Simulation Manager* talks directly to the Node/Network manager in order to coordinate network activities (DIS players). The *Simulation Manager* module also coordinates activities between the Playerset (Main application) and the Simulation Synchronization. This module is the main driver of the simulation and gives authority to the application to execute events.

3.3.11 Node/Network Manager. The node/network manager is the main interface between the hardware layer and the DIS interface. It will filter incoming DIS messages to determine if the respective DIS player is associated with the LP on which the manager is residing. It is also the responsibility of the node/network manager to pass external events to the DIS manager so DIS players are updated based on the actions of the simulation players. A generic interface allows calls to several types of hardware independent communication protocols.

3.4 Hardware Layer

This section contains the operating system specific system calls in order to obtain such items as the system time. This layer is also the main interface to the selected communication protocol used between processors in distributed systems, or between nodes in parallel machines. Proposed platforms include the Intel Hypercube, Paragon, and a network of Unix platforms using either an MPI or PVM message passing scheme.

3.5 The Application

An expanded generic representation of the application model showing BattleSim specific players is shown in Figure 11. The relevent BattleSim specific features include the types of events (excluding DIS events) and the fact that a hierarchical player is a vehicle. The following subsections describe the proposed interaction of the modules within the application.

3.5.1 Playerset. Playersets contain all players on an LP or processor. Three subclasses will inherit basic features from the generic playerset. The playerset may be comprised of local application players, copies of these players from another LP or representations of DIS players. These appropriate playersets will change dynamically based on movement of a player (in a spatially partitioned simulation) and instantiation or destruction of a player.

3.5.2 Player. The player module is a simple data storage device for basic descriptions of players. This is a generic player and can be adapted for any simple player from any domain in which players move. For domains in which players do not move or have characteristics matching those listed in Table 1, the values listed can be set to *null* and the subclass characteristic can be

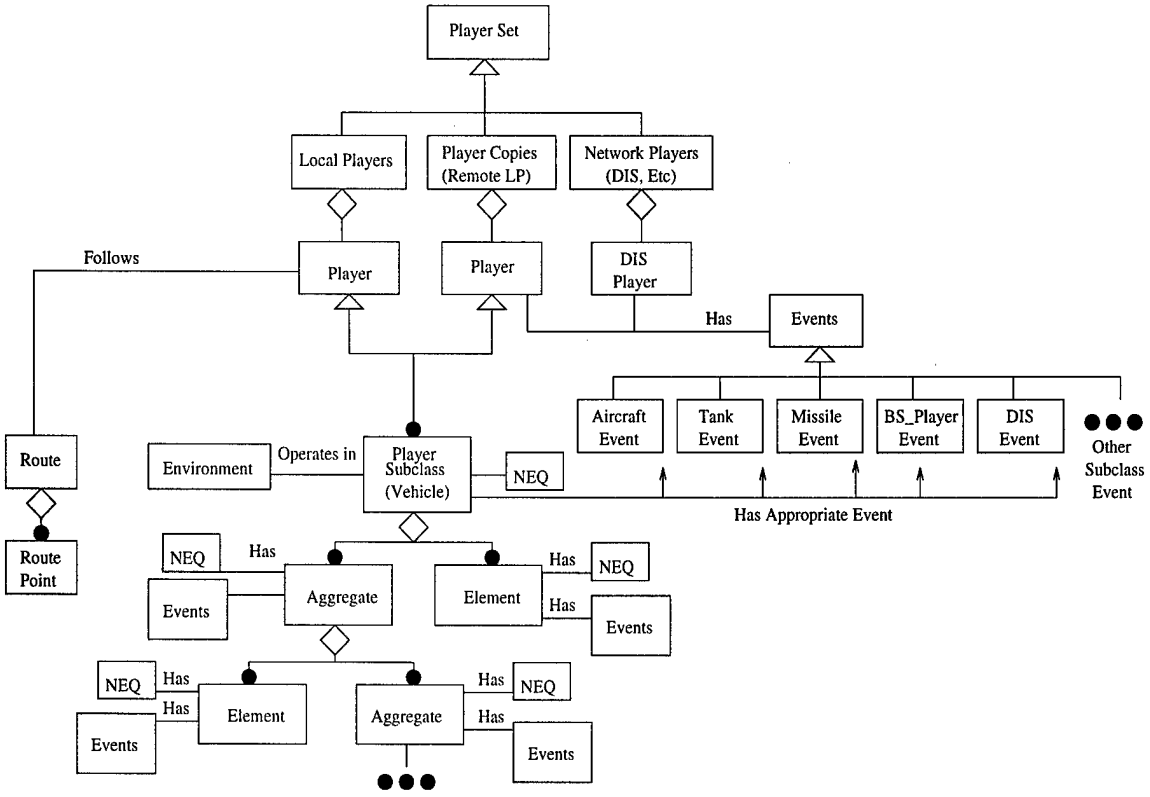

Figure 11. Application Layer

Table 1. Attributes of a Generic Player

Attribute
object_type
object_id
current_time
num_events
NEQ
#components
components
x_position
y_position
z_position
x_velocity
y_velocity
z_velocity
roll
pitch
yaw
roll_rate
pitch_rate
yaw_rate
player_size
mass
subclass

used to insert new domains. The player module includes operations to allow the manipulation and retrieval of player data.

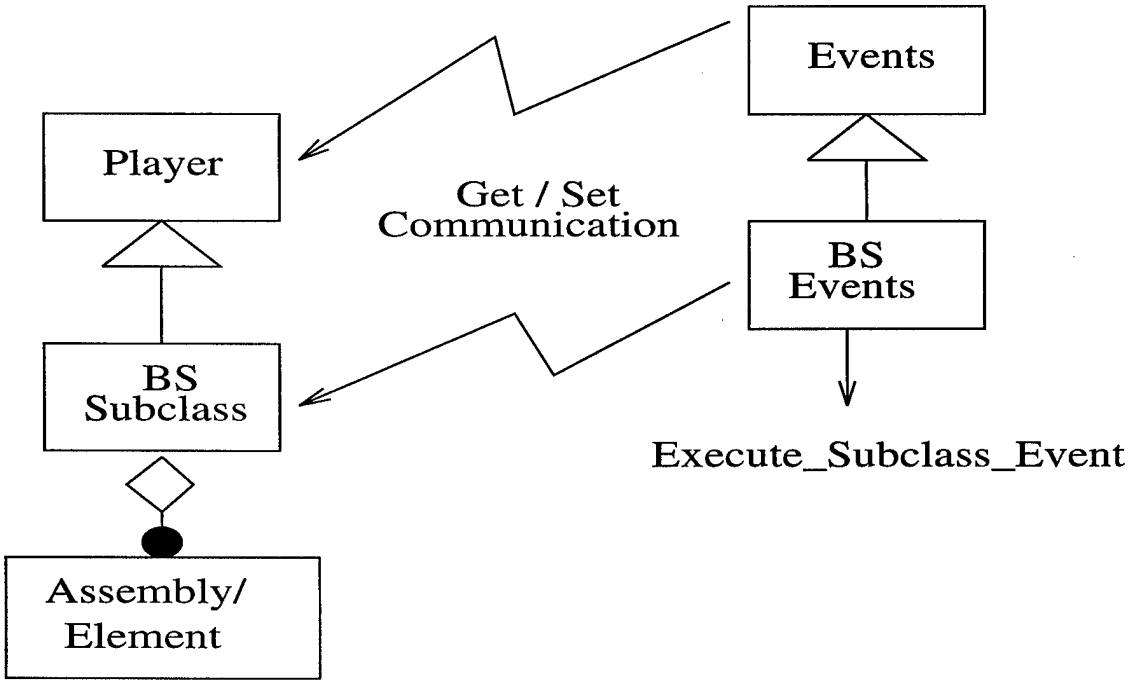

Figure 12. Event Representation

3.5.3 Events. Event types may be similar among players (such as turn, destroy object, etc.) The *simulation manager* will call the execution of the events from this section of the application. The event code provides methods for the events which will access the player to gain required information in order to perform necessary calculations. A function *execute_subclass_event* will pass unknown events to the next lowest subclass event handler in the hierarchy. The event class will also handle event prediction and scheduling. A representation of this description is shown in Figure 12.

3.6 The Hierarchical Player

3.6.1 Vehicle. As shown in Figure 11, the vehicle is a subclass of the player class. This module inherits all attributes from the player class and can add more desired attributes to expand the domain. For example the BattleSim player can be inserted in place of this module in order to add BattleSim functionality. The BattleSim player, as is exists now, must be modified with a function to call assemblies and elements which may reside in a hierarchy below this class. Whichever domain subclass is used in this position, it must have its own event handler with a function to call its assemblies and elements within its hierarchy. This subclass inherits the attributes of the player event handler. This concept is shown in Figure 12.

3.6.2 Assemblies/Elements. Each Assembly or Element will maintain its own NEQ and event handling modules. The assembly/element NEQ uses the same NEQ class used by the simulation system. However a new instantiation must be made in order to guarantee separate NEQ's. Section 3.11 discusses the interaction of assemblies and elements in an example, and Figure 18 shows a graphical representation of this section. Two approaches are recommended for the communication between assemblies, elements and players when two or more components have dependent event prediction. One is a parent to child relationship where assemblies and elements communicate in a tree like structure passing messages only between parent to child or child to parent. The second approach is to use "flattening" so that assemblies and elements may talk directly if not in the same tree structure. Under the parent to child communication scheme, the child will be responsible for informing its parent of internal events so they eventually will be logged in the LP NEQ so that proper synchronization may occur. The first approach will be used for ease of testing and implementation.

3.6.2.1 Environment. As in the JMASS model, the environment is treated as a player. The environment contains boundary information and player location. The environment is also in control of predicting the interaction between players in an application. The environment should be able to be expanded to include terrain and weather information which may affect the simulation at a later date. The simulation is also aware of all the types of players in an application in order to enforce the rules determined for the environment. For example, if an airplane and a cloud were modeled in an application, both would be players and would not be able to collide (and cause damage).

3.7 Initialization of Simulation

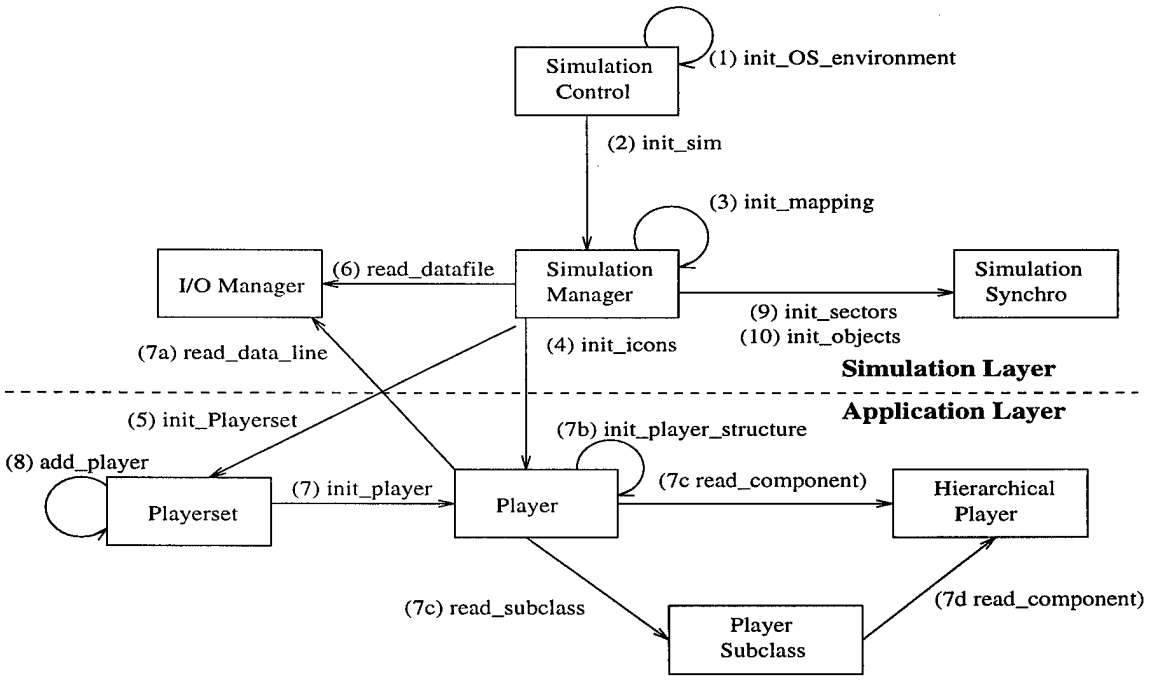

Figure 13. Simulation Initialization

Figure 13 shows the initialization of the simulation layer and its interaction with the application layer. The order of initialization is labeled within Figure 13 and follows a sequential order.

This order is described in functionality throughout the rest of this section. The *simulation control* module calls the function *init_OS_environment*; this function initializes the distributed/parallel environment as described in Section 3.3.1. Once the distributed/parallel environment is initialized, the *simulation control* module calls the function *init_sim* in the *simulation manager* module and is not returned to until the end of the simulation. The *simulation manager* calls the function *init_mapping* which calls the *player* module in order to determine player types and player interaction events. This information is primarily used by the *simulation synchronization* module during the simulation. The function *init_mapping* then calls the *init_icons* function to get the player type information.

The *simulation manager* module then calls the function *init_playerset* in the *playerset* module in order to initialize the linked list which will store the application player information. This storage process is discussed in Section 3.9 in more detail. The *init_playerset* function in-turn calls the *init_player* function residing in the *player* module until all players have been read into the simulation (iterative loop). The *player* module then calls the *I/O manager* module in order to read the player data from the scenario file.

The *player* module then calls the *init_player_structure* function which determines either to call *read_subclass* or *read_component* based on the player type. Further discussion of the initialization of the hierarchical player is discussed in Section 3.8. The *player* module then passes the player information back to the playerset so that the *playerset* can call the function *add_player* to update the playerset data structure.

The *playerset* then returns control to the *simulation manager*. The *simulation manager* then uses information it read previously to determine what type of partitioning will be used and will

call either or both of the functions *init_sectors* and *init_objects* based on the partitioning type information. The *simulation manager* then proceeds to call the function *start_sim* to get the simulation running. This process is discussed in Section 3.10.

3.8 Hierarchical Player Initialization

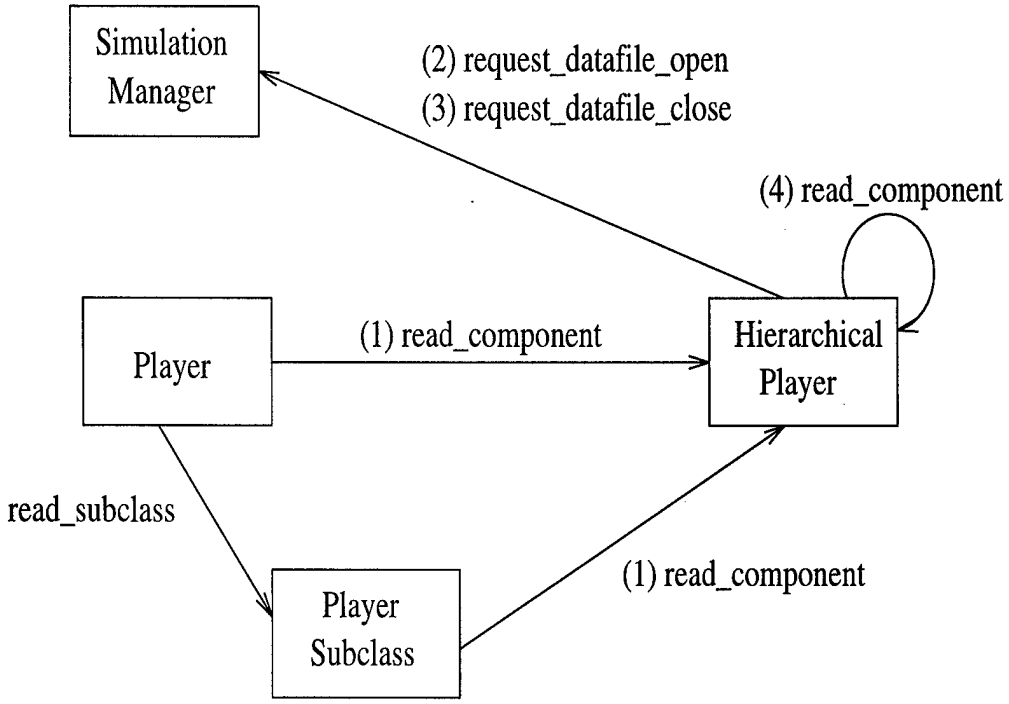

Figure 14. Initialization of Hierarchical Players

As shown in Figure 14, a hierarchical player is initialized. The rest of this section describes in detail of how this diagram is executed in the new simulation system. From Section 3.7, once the player type is identified it either calls *read_component* or *read_subclass*. If *read_subclass* is called, it may call *read_component* based on the player type. The component in the hierarchical player then calls the *request_datafile_open* in the *simulation manager* in order to open the datafile associated with the hierarchical player. Once the data file is opened, and the data is read, the function *request_datafile_close* is called to close the datafile. If the component type is identified to have

other aggregate components, *read_component* is called again and the process is repeated until the hierarchical player is completely read. Once the hierarchical player is finished initializing, control is then returned to either the *player subclass* or *player* depending on which module initially called for initialization of the hierarchical player.

3.9 Storage of Hierarchical Players

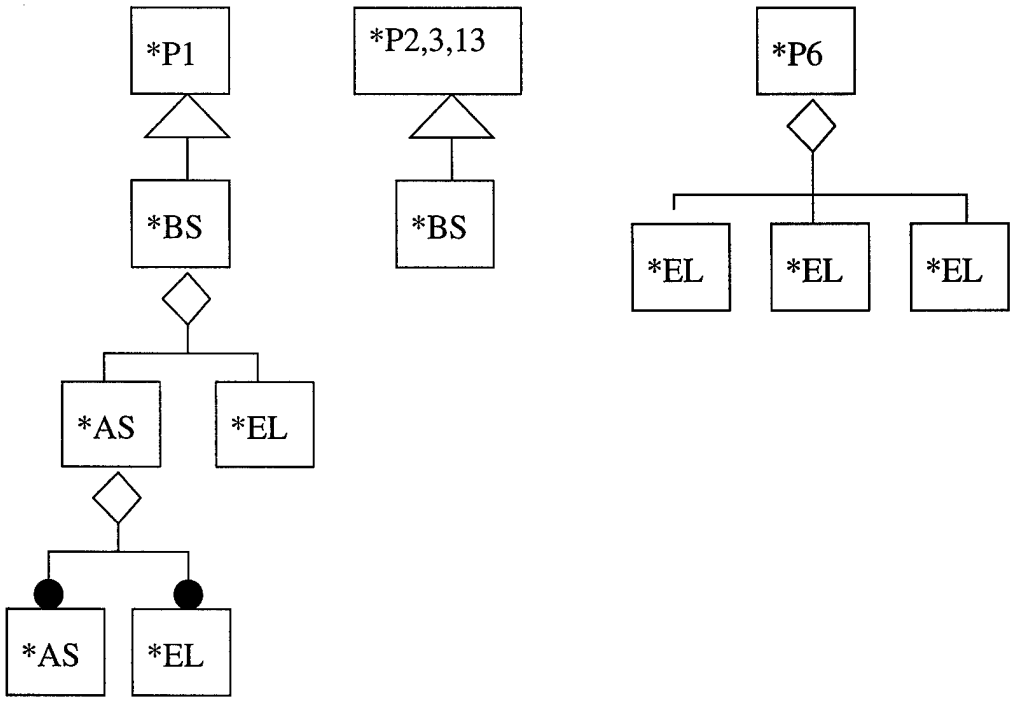

Figure 15. Examples of Hierarchical Players

The storage of hierarchical players is of interest to form a standard for programmers of new hierarchical players for this simulation system. This simulation system is aware of the entities *playerset* and *player*. These are generic representations which can be applied to nearly any domain since they contain basic information, and subclasses can be used through inheritance to expand this model. Figure 16 shows the storage of multiple types of players as shown in Figure 15. Realizing that there are multiple types of storage systems and search algorithms available, Figure 16 shows

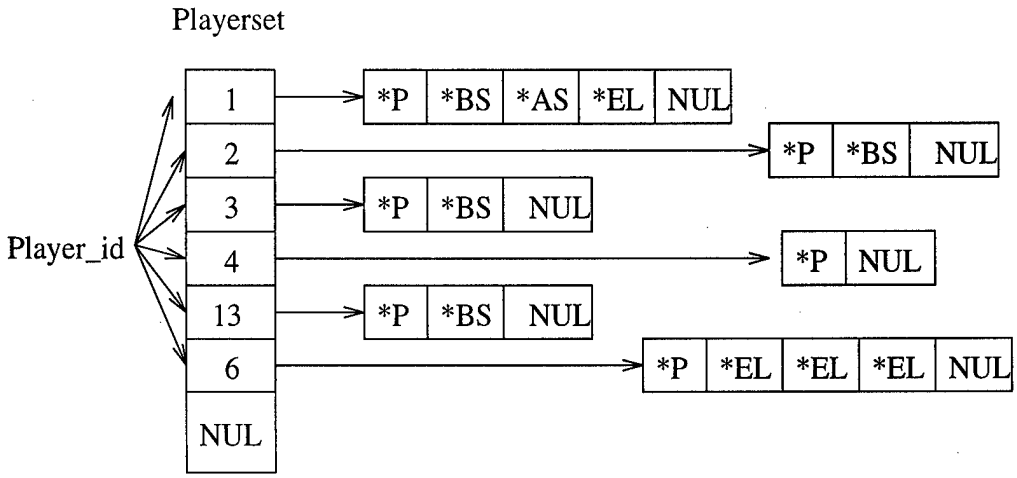

Figure 16. Storage of Hierarchical Players

a representation for implementation (6). The implementation must be standardized so that if a KBSE system is used in conjunction with this simulation system, data is stored in the proper format. Section 3.3.2 mentions one type of a KBSE that can work with this system. In Figure 16 the playerset is represented as an array which contains pointers to a linked list. The following notations are used in this diagram:

- *P(Player_ID) (Generic Player Class)
- *BS (BattleSim Subclass)
- *AS (Assembly: Hierarchical Player component)
- *EL (Element: Hierarchical Player component)

The storage of these players in this linked list will be implemented in the *init_player_structure* function in the *player* module as shown in Figure 13. Hierarchical players can be represented as a tree of objects, so a simple search algorithm such as *depth-first-search* can be implemented to store these objects in a methodical order. In Figure 16 player_id 1 shows a simple player with a

BattleSim subclass and two hierarchical components, an assembly and an element. Player_ids 2, 3 and 13 show a BattleSim player; player_id 4 shows a simple player with no subclasses and player_id 6 shows a simple player with three elements. From these linked lists, the actual player structure is unknown, relying on the storage and retrieval search algorithm to determine structure. This is shown in Figure 15 because the additional assemblies and elements are actually tracked by the assembly that owns them. A minimum requirement for this simulation system is to have a generic player class. Hierarchical components cannot exist without this basic player class.

3.10 Simulation Execution

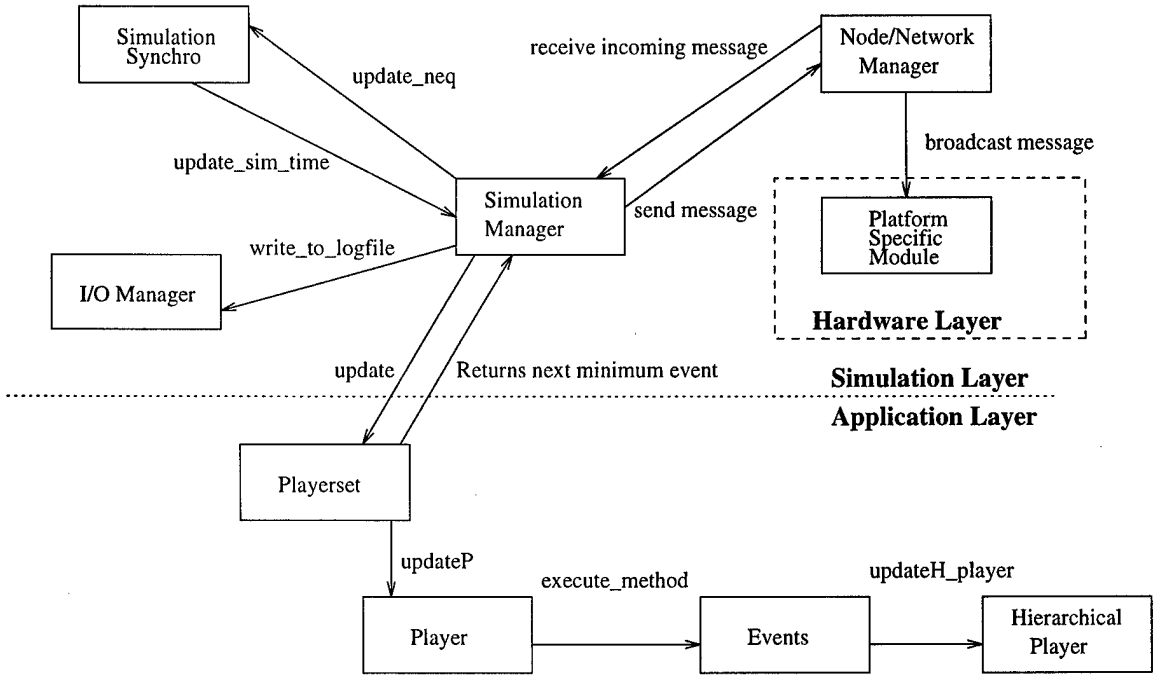

Figure 17. Simulation Execution

Figure 17 describes the iterative action of simulating events in this discrete-event simulation. This section describes the interaction and function calls used between the simulation, application and hardware modules. Assuming a conservative synchronization approach, the *simulation manager*

receives messages from all incoming communication channels via the *node/network manager*. These messages indicate the minimum time for interaction between players on different LPs to interact. In an optimistic approach this message would most likely be a rollback command.

Once all messages are received and it is determined by the synchronization algorithm that it is safe to proceed to a safe time τ, the simulation manager gets the next time ordered event from the *NEQ* and calls the *playerset* module, on that LP, with the appropriate player information to include the *event* and the *player_id*. The *playerset* then calls the *updateP* command in the *player* module. If the event passed with the update data is a generic player call, then the event is executed, a new player event is predicted, and the next predicted event is returned. Discussion of how a hierarchical player follows this event flow is discussed in Section 3.11. Once the *simulation manager* receives the new event, the new event is updated in the processor NEQ. The *node/network manager* is then called to make any specific synchronization calls to the other participating processors and to convert the simulation event into a DIS PDU packet to be broadcast to other DIS players via the platform specific module. The *I/O manager* is then called to write the *event*, *player_id* and *simulation time* to the log file. This process repeats until the simulation end time is reached.

3.11 Hierarchical Player Execution

As described in Section 3.10 the generic player *event* module calls the next hierarchical player as defined from the playerset module storage mechanism discussed in Section 3.9. Each Hierarchical component is aware of all of its children and who its parent is. Stepping through the example shown in Figure 18, *Assembly #1* does not recognize the event as its own, so it passes the event to its children, *Assembly #2* and *Element #3*. *Assembly #2* does not recognize the event so it passes it to its children, *Element #1* and *Element #2*. *Element #2* recognizes the event, executes the

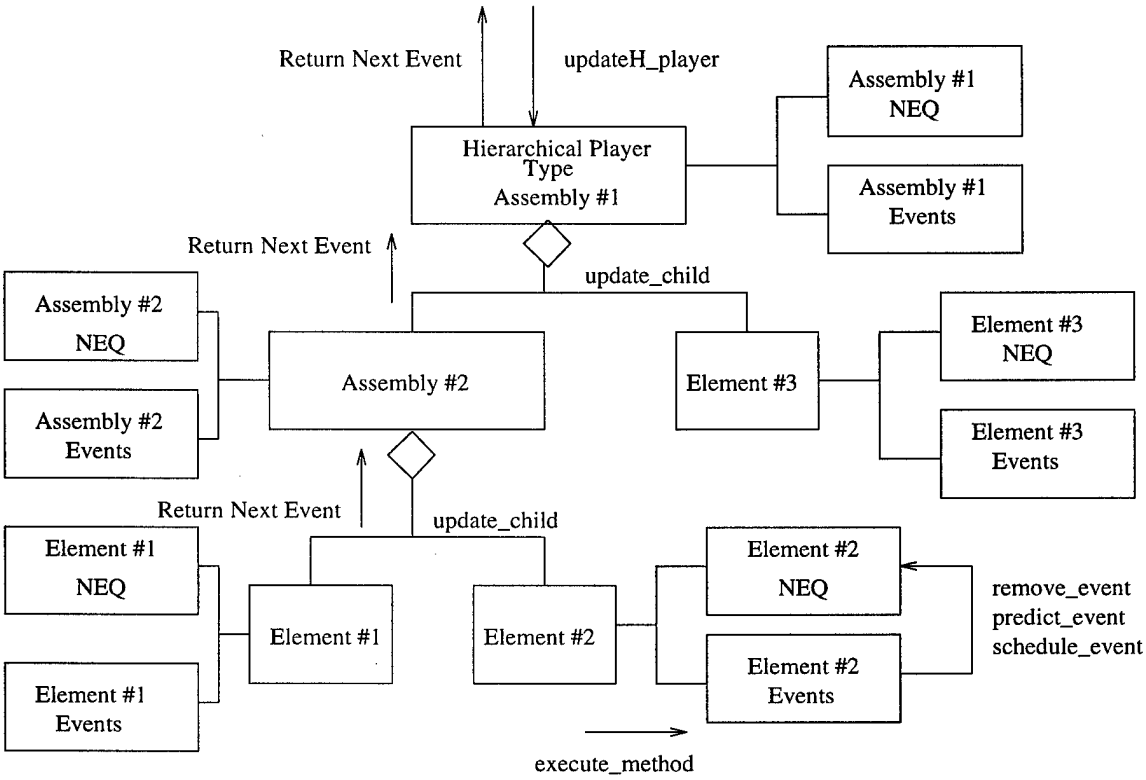

Figure 18. Hierarchical Player Execution

event, removes that event from the *Element #2 NEQ*, predicts a new event, schedules the newly predicted event on the *Element #2 NEQ* and returns a status message of the new event and the event simulation time to its parent. This information is passed to the top of the hierarchical player until it can be handled by the *simulation manager* as described in Section 3.10.

3.12 Portability

Provided this design is implemented in a language which is compilable for the desired platform, there are only two modules which need to be re-written in order to port the implementation of this design to other hardware platforms. These are the *simulation control* module and a module residing in the hardware layer for the appropriate platform. These modules are shown in Figure 10.

3.13 Summary

This chapter discusses the design and interaction of the simulation system with the application layer. Where appropriate, event flow diagrams are provided in order to show how these modules work together.

IV. Analysis and Building of Simulation Architecture

4.1 Introduction

One of the main considerations during this project was to maintain the functionality of the existing BattleSim program while making modifications to the existing code to allow modularity and portability for future versions. This chapter focuses mainly on converting the original BattleSim into the new simulation architecture while allowing for code restructuring to add further capabilities such as modular hierarchical players, hybrid (object/spatial) partitioning schemes and software portability to other platforms which will allow parallel/distributed processing of the new Simulation architecture.

4.2 Mapping of Old BattleSim to the New Architecture

As stated before, reuse was a necessary part of maintaining the integrity of the original BattleSim functionality while also allowing for better object representation for the modularity of the software components involved. During the reuse analysis, code in the original BattleSim was removed in order to aid in better readability of the code. The old history records were removed since this information is archived. Numerous "IFDEF" statements were also removed since this code was identified as test statements which were constructed by the code designers at the time of development in order to test the proper functionality of the code they had written. Since no original specifications from a software engineering perspective were available with this code, verification of the code modules could not truly be conducted and it was assumed that the code presented works as conceived by the original programmers.

The following subsections describe the reuse of the original BattleSim code in order to build the new modules. These subsections are broken into three categories in order to reference their appropriate design diagrams. These categories are: simulation layer, hardware layer and application layer. This layout follows a layered approach as discussed in Section 3.2. The code itself is written in the C programming language due to the fact that the reused code was written in C. The target platform for this implementation is the Intel Hypercube since the original BattleSim code was written for this specific platform. Tables showing original BattleSim modules which were divided to form the new architecture are listed in Appendix B.

4.2.1 Simulation Layer.

4.2.1.1 Simulation Control. The simulation control module is a direct adaptation from the original BattleSim's "host3.c" program. The behavior of this module matches the described behavior of the *simulation control* module as designed in Section 3.3.1. The host3.c program initializes the appropriate number of nodes on the hypercube and sets up the required communications between the nodes. It then calls the BattleSim module "battle.c". This was modified so that it invokes the *simulation manager* in the new simulation architecture.

4.2.1.2 I/O Manager. The I/O manager module is a direct implementation from the "sim_read.c" and "interfaceB.c" module in BattleSim. Although the design does not account for interaction between the I/O manager and the application, this does not seem to be a significant problem since the I/O module is only used during initialization of the simulation when the I/O module uses the call *read_player*. In-turn, all players use the function "read_data_line" from the I/O manager in order to read associated data to fill the player with data. This allows a standard

call to the I/O manager that is not BattleSim specific, thus allowing other domains to follow these two basic requirements. Functionality for *VISIT* logging was also added to the I/O manager.

4.2.1.3 Simulation Manager. The simulation manager uses functions from the modules listed in Table 2. The second column in Table 2 shows if all the functions (total) or a subset (partial) were used. Appendix B contains Table 7 which lists all of the BattleSim functions which were used from these modules.

Table 2. Reused Modules for Simulation Manager

battle.c	partial
interfaceB.c	partial
object_manager.c	total
tchmap.c	total
sim_drive.c	total
sim_cntrl.c	total

4.2.1.4 Node/Network Manager. The node/network manager module in the new simulation architecture uses components from the original BattleSim module "interfaceB.c". The calls used pertain directly to the calls "interfaceB.c" makes to "lpman.c" and "cube2.c". The functions which were reused from "interfaceB.c" are listed in Appendix B in Table 8.

4.2.1.5 Partitioning filter. The partitioning filter module is composed of functions from the module "sector.c" in the original BattleSim. This sets up the rules for a three dimensional spatial partitioning scheme. The original BattleSim did not incorporate a method for object partitioning.

4.2.1.6 Spatial. The spatial module is composed of the sector rules for orientation from the original BattleSim code "sector.c".

4.2.1.7 Clock. The clock module is a direct implementation of the TCHSIM module "clock.c".

4.2.1.8 NEQ. The NEQ module is a direct implementation of the TCHSIM module "neqA.c". However, this module does not support instantiation of multiple instances of this class. Section 4.4.1 describes the changes made to meet the design specifications for the NEQ as discussed in Chapter III.

4.2.1.9 Event. The event module is a direct implementation of the TCHSIM module "event.c".

4.2.1.10 Simulation Synchronization. This module uses components from the BattleSim module "InterfaceB.c" which are specific to its lower level components. These calls include the interface to the TCHSIM modules *clock*, *event*, and *neqA*. The functions used from "interfaceB.c" are listed in Appendix B in Table 5. This also violates the original design because the player and event predictor module in the original BattleSim depend on functions residing in "interfaceB.c". The solution to this dependency is to add an additional call which interfaces with the calls associated with these three modules into the *simulation manager*. This adds overhead during each event execution which can be seen in the results listed in Table 4.

4.3 Hardware Layer

As stated before, this implementation used a significant amount of re-used code from the original BattleSim and TCHSIM. In order to verify the correct operation of the new architecture in comparison to the original BattleSim, the selected platform for this implementation was the Intel Hypercube.

4.3.1 Hypercube. This module in the new architecture is adapted from the original BattleSim code modules "cube2.c" and "lpman.c". The direct correlation of the "cube2.c" module is straight forward since this module was already written for implementation on the hypercube. The implementation of "lpman.c" was decided upon due to the fact that "lpman.c" uses UVA spectrum formats for message passing. This union of code modules to adapt to the new architecture is in compliance with the design proposed for the hardware layer as defined in Section 3.4.

4.3.2 Application Layer.

4.3.2.1 Player Set. The playerset module is a direct implementation from the original BattleSim code "playerset.c".

4.3.2.2 Player. The player module is a direct implementation from the original BattleSim code "player.c". However, the player structure has additional attributes added to support the hierarchical players. Section 4.4.2 discusses the new structure of the player module.

4.3.2.3 Events. The events module in the new architecture is a union of the original BattleSim code to include modules listed in Table 3. Functions from these modules are listed in

Appendix B in Table 10. However, additional function support must be implemented to support the hierarchical player. This support is discussed in Section 4.4.3.

Table 3. Modules Used in Events

predict.c
schedule.c
ex_event.c
methods.c

4.3.2.4 Route. The route module is a direct implementation from the original BattleSim code "route.c".

4.3.2.5 Route Point. The route point module is a direct implementation from the original BattleSim code "rt_pt.c".

4.4 Implementing Support for Hierarchical Players

The original BattleSim code does not allow support for hierarchical players. Modifications to the modules listed in the following subsections must be made in order to support the hierarchical player structure.

4.4.1 NEQ. The current NEQ does not allow for instantiation of a new NEQ in the function *Xinit_neq*. This should be modified so that it returns a pointer to the module that calls it. Functions which update the NEQ should be modified to pass a pointer reference to which NEQ they wish to access.

```
struct playerclass
{
        int object_type;
        int object_id;
        double current_time;
        int update_origin;
        int num_events;
        void *NEQ;        /*pointer to the NEQ*/
        int num_components; /*number of components*/
        array [*component]; /*an array storage of
                              pointers to direct
                              hierarchical components
                              of the player class */

        struct location_type location;
        struct xyz_velocities velocity;
        struct orientation_type orientation;
        struct rotation_rates rotation;
        double player_size;
        double mass;
        void *polygon_list;
        void *subclass;
};
```

Figure 19. Player Data Structure

```
struct componentclass
{
        void *parent;          /*pointer to parent*/
        void *NEQ;             /*pointer to the component NEQ*/
        int num_components;    /*number of components*/
        array [*component];    /*an array storage of
                                 pointers to direct
                                 hierarchical components
                                 of the player class */
        int component_type;    /*definition of the
                                 component type
                                 used for reading
                                 aggregate components*/
         /*additional
           component
           data fields
           specified
           for this
           component*/
};
```

Figure 20. Component Data Structure

4.4.2 Player. Figure 19 shows the proposed data structure to use. This data structure will allow support of the hierarchical players as defined in Chapter III. Storage of the hierarchical components is described in Section 3.9.

4.4.3 Events. Currently as implemented, the player's event module recognizes unknown events and prints an error message. In the case of an unknown event, this recognition of unknown events should call the *execute_component_event* which will in-turn call the appropriate component call based on the definition of the event. Methods to include the retrieval of subclass data structures are also contained in this module. This is due to the lack of inheritance capability in the C programming language.

4.5 Implementation of Hierarchical Players

4.5.1 Component Class Requirements. Each component will have the minimum attributes as shown in Figure 20. It will also provide the attributes associated with the specification of that particular component. Methods in the component class will include *get* and *set* calls for each attribute which will be accessible by the *component event handler.* Elements will have zero components and assemblies will be constructed of one or more lower level components.

4.5.2 Component Event Class Requirements. Component event class requirements are required to provide methods for event execution for the component, event prediction, event scheduling for the component NEQ and a method to update lower level components.

4.6 Hybrid Partitioning Schemes

The original BattleSim program only allowed for spatial partitioning of the simple players. This was done by using a three dimensional partitioning module called "sector" in the original BattleSim. The concept of object partitioning was not built into the original BattleSim. In order to accomplish this, three items need to exist. The first required item is a module which maps players and player components for hierarchical players to a specified LP on the hypercube or processor in a distributed system. This was done by creating the *object* module shown in Figure 10.

The second requirement was to add a designator to the player class to identify it as either a simple player or a hierarchical player. The difference between a simple player and a hierarchical player is that a hierarchical player will have an extra data structure filled with hierarchical player information which will identify the number of direct children this player has and the simple player's data structure will be filled with NULL statements. Figure 16 in Chapter III shows three various configurations of players to include: a simple BattleSim player, a hierarchical player and a simple

player with no subclasses. As discussed in Chapter III, a parent to child communication process is only allowed at this time and may be expanded depending on the design of the model.

The third requirement is to modify the original BattleSim data file in order to indicate a player which is either spatially or object partitioned.

4.7 Simple BattleSim Players and Hierarchical Player Interaction

Whether a player is simple or hierarchical, the same interactions will occur between the players. These basic interactions as defined are *collision, attack* and *sensor detect.*

At this point in time, interactions of hierarchical components are completely transparent to the user. For example: if a radar array (a component on a hierarchical player) were to detect an enemy aircraft it will be treated as if the hierarchical player detected the enemy plane and not the sensor array.

4.8 Portability Issues

As shown in Figure 10, there are two modules which must be recompiled into the simulation layer when changing platforms. These include: Simulation control and the selected module for the desired platform (i.e. Sun, Paragon, Hypercube, etc.). As designed, all hardware interaction is done through the appropriate module for the desired platform. The Simulation control module, as it currently exists, is the module "host3.c" as shown in Figure 8. In "host3.c", a host is set up on the hypercube to interact with the LPs. The module "cube2.c" as shown in Figure 8 contains all of the required calls to the hypercube's operating system.

When porting this "C" code to other distributed platforms, such as a network of Suns, the MPI or PVM (or a similar message passing interface) should be constructed and contained within the "Sun" module and the "Simulation Control" module should be adapted to initialize the desired protocol on the distributed network before initialization of the simulation occurs.

4.9 Conclusion

This chapter focuses mainly on the reuse portion of the old BattleSim program and describes methods to adapt to the new architecture. Issues faced by the new architecture are also addressed. These include interaction of simple and hierarchical players, and portability issues.

V. Test Results of the New Architecture

5.1 Introduction

This chapter focuses on the test results of the new simulation architecture using the BattleSim subclass in comparison to the original Battlesim architecture using the same scenario to ensure proper functionality between the two simulation systems.

5.2 Original BattleSim VS. the New Architecture

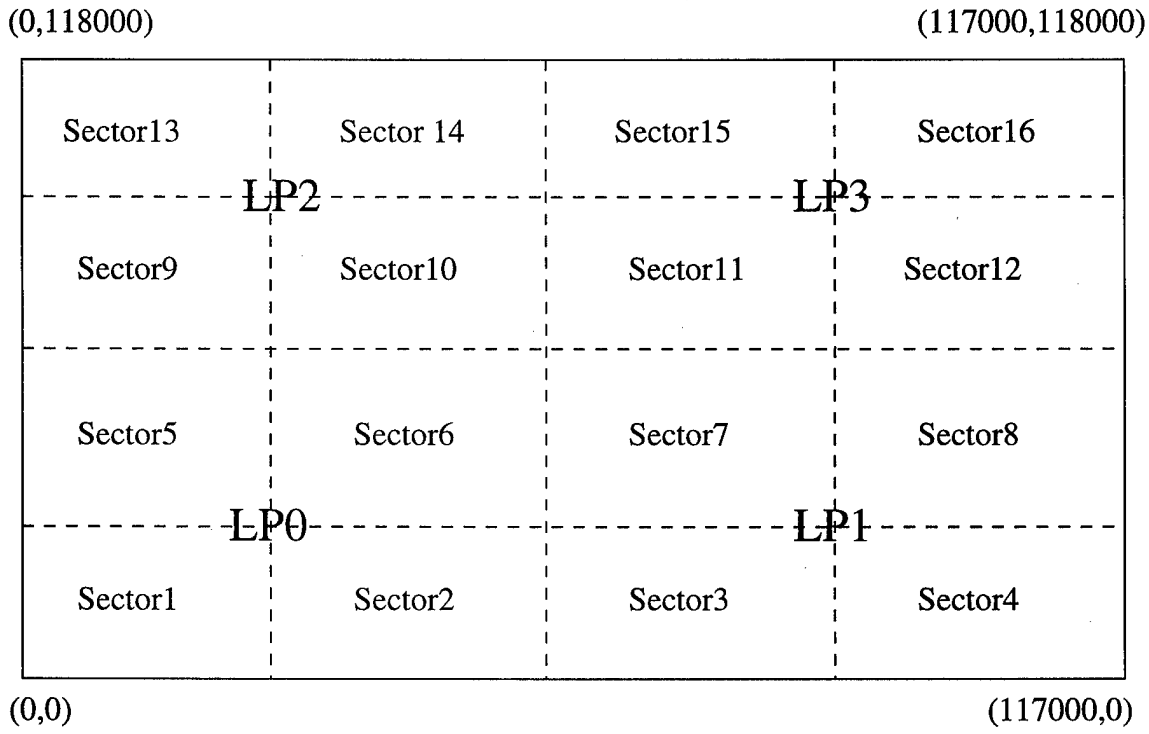

Figure 21. Scenario Battlefield

In order to verify that the original BattleSim scenario files can be used with minor modification and correct operation, three test cases were executed by both systems and compared. The scenarios are discussed on a case by case basis in the following subsections. Figure 21 shows the layout of an example battlefield which has sixteen sectors and four LPs.

Table 4. Test Results: Original BattleSim vs. New Architecture

Architecture	#LPs	#Players	Simtime	Realtime (sec)	%Diff
New Architecture	1	1	10000	135010	N/A
BattleSim	1	1	10000	N/A	N/A
New Architecture	2	4	60000	185183	133
BattleSim	2	4	60000	138514	75
New Architecture	8	40	1000	716083	196
BattleSim	8	40	1000	363499	51

Figure 21 shows the layout of the battlefield. Output files were compared to verify correct operation of the new architecture using the original BattleSim output file as a benchmark.

5.2.1 Test Case 1: Sequential Operation. The sequential version of this test case was written to verify correct operation of the new architecture on a single processor. This test case removes the element of parallelism in order to aid in debugging. The sequential scenario was written with Hiller's *BSGen* scenario generator (10). The sequential scenario did not work with the original BattleSim application. Existing sequential scenarios were tried (Bench21 w/1LP) to no avail with the original version of BattleSim. Event execution consisted primarily of reaching turnpoints in the scenario and were verified against the original input file. No comparison between the architectures could be made due to lack of results from the original version of Battlesim. Results are listed in Table 4.

5.2.2 Test Case 2: Bench21. This test case was designed in order to verify that BattleSim players are properly replicated when entering a sector owned by another LP . This scenario uses one BattleSim object (a plane) and 8 sectors divided between two LPs. This scenario was designed to test correct execution of events when players cross sectors in a zig-zag pattern. The plane flies in a zig-zag route in the x direction and then a zig-zag pattern in the y direction. As shown in Table

4, The new architecture required 1.33 times longer to finish the scenario. This is probably due to the amount of overhead in limiting calls to the new architecture which allows a larger application base. Output files were sorted by LPs and compared between the two programs to verify correct operation of the new simulation system.

5.2.3 Test Case 3: scen96a. This test case was designed by Hiller in order to test a new synchronization scheme. This scenario contains 40 players and uses 64 sectors on eight LPs. As shown in Table 4, the simulation time required for the new simulation architecture to complete is almost double the time that it required the original BattleSim program to complete. This is again related to the overhead required to strictly define the interface between the simulation layer from the application layer which will allow applications from different domains to be used with this simulation system. As in *Test Case 2*, output files were sorted by LPs and compared between the two programs to verify correct operation of the new simulation system.

5.3 Summary

Verification of the new simulation architecture using the BattleSim subclass was successful. This will allow tools such as Hiller's *BSGen* program to be used for BattleSim type simulations with the new simulation architecture.

VI. Conclusions and Further Research

6.1 Summary of Results

Problems with reuse caused a delay in the forward progression of the implementation of the design specified in Chapter III. These reuse problems are discussed in detail in Section 6.1.1. However the new architecture successfully supports the functionality of the original BattleSim program using a BattleSim subclass. The results indicate that, due to the increased overhead to tightly couple the interface between the simulation layer and application layer, communications are increased by a factor of two based on the simulation times reported in Table 4.

6.1.1 Reuse Problems. The whole premise of using an object oriented approach to design and implement a system is to be able to define specific interfaces for coupling of models. The problems encountered during this thesis effort regarding the reuse of code can be accounted for by the fact that the BattleSim code was not written to be re-used. This was a simulation system whose only purpose was to simulate a battlefield environment. This led to complications when trying to reuse the simulation related portion of this code because this portion of the code relied heavily upon the application code.

The reuse philosophy of software engineers in general is that *no* modifications should be made that affect the operation or functionality of the code. Modifying the code in any way may introduce unexpected bugs in the software. In an attempt to modify multiple functions in order to reduce dependencies between the existing modules, I had to devise a new approach to reuse which allowed modification to the dependency of the code to fit the new architecture as designed. The first attempt at this approach failed and severely limited the forward progress of the code. A second approach corrected the problems encountered during the first round of programming, but from

the experience encountered it only verifies the fact that when reusing existing code, modifications should not be made to the code. In retrospect the code that needed to be modified should have been written from scratch. This may have reduced the time required to finish this project. This, however, might have not conformed to the operation of the original BattleSim.

6.1.2 Performance. As shown in Table 4, the new architecture executes slower than the original BattleSim while using identical scenario files. This is most likely due to the restructuring of the code causing additional layers of calls to define a strict interface to the application.

6.1.3 Attainment of Goals. As discussed in Section 1.2, there were two specific goals of this research. The attainment of these goals are discussed in the following sections.

6.1.3.1 Simulation Architecture. A simulation architecture was designed to support multiple hierarchical players. The main simulation engine was implemented and the design for the hierarchical players was defined in Chapter III. Due to time limitations the hierarchical player was not implemented or tested with the new simulation system. Section 6.3.1 discusses what additions to the existing structure need to be accomplished to meet this goal.

6.1.3.2 Interface to DIS. The actual design of the interface to the DIS standard was not completed. This was postponed in order to focus efforts on the design an implementation of the new simulation architecture.

6.2 Research Contribution

This research produced a reusable simulation system in which many applications can be written. This will allow further research in areas other than battlefield simulation. With a well

defined interface between the simulation and the application level, the simulation system does not have to be modified in order to create new applications. This design integrates some of the best features available in object-oriented hierarchical simulations. Written in an object-oriented style and in the C programming language, this model can be easily ported to other parallel or distributed environments without much effort in order to perform additional comparisons between platforms.

6.3 Recommendations for Further Study

6.3.1 Remaining Work. Following designs for the interaction of the hierarchical player discussed in Chapter III, and the partial implemementation discussed in Chapter IV, the following items need to be accomplished in order to make this system complete.

- Modify the neqA program to have an instantiable NEQ.
- Expand the attributes of the player class data structure as discussed in Chapter IV.
- Program the assemblies and elements of a hierarchical player based on the data structure shown in Figure 20.
- Add the event handler to update hierarchical players in the basic player's events module.

6.3.2 Graphical Simulation Editor. Possible future research into the simulation construction through the use of a graphical Knowledge Based Software Engineering (KBSE) approach may be done with this module. This would allow the use of a graphical interface which accesses a database of simulation objects in order to design a simulation through software specifications which were previously designed for the desired domain to be simulated.

6.3.3 Optimization Algorithm. Design and implement an algorithm which analyzes a KBSE library of components to conclude the best possible hybrid partitioning scheme (spatial and object) based on the functionality of the KBSE components.

6.3.4 Optimistic Synchronization. Design and implement an optimistic synchronization approach to the simulation. Once this is done, hybrid synchronization schemes may be designed and implemented.

6.4 Summary

This research project investigates a parallel discrete-event simulation system which allows the use of hierarchically constructed players. The architecture was implemented in the "C" programming language for the Intel I386 eight node Hypercube using the UVA SPECTRUM message passing scheme.

Appendix A. Definitions and Acronyms

- **Assembly**: An aggregate formed by one or more elements or aggregates
- **Causal**: The time ordering of events in sequential increasing order.
- **DEVS**: Discrete Event System Specification
- **Digress**: The route to the origin from the target
- **DIS**: Distributed Interactive Simulation
- **Egress**: The route to a target from the origin
- **Element**: A low level component
- **Hierarchical**: The ordering of objects in a tree-like structure
- **JMASS**: Joint Modeling And Simulation System
- **LP**: Logical Process
- **NEQ**: Next Event Queue
- **OCU**: Object Communication Update
- **SSM**: Software Structural Model

Appendix B. Tables of Reused Code

Table 5. Reused Functions for Simulation Synchronization

New Module	Function	Battlesim Source
Simulation Synchronization	show_neq	interfaceB.c
Simulation Synchronization	record_neq_state	interfaceB.c
Simulation Synchronization	restore_neq_state	interfaceB.c
Simulation Synchronization	show_neq_state	interfaceB.c
Simulation Synchronization	clear_packed_neq	interfaceB.c
Simulation Synchronization	add_event	interfaceB.c
Simulation Synchronization	count_event	interfaceB.c
Simulation Synchronization	neq_time	interfaceB.c
Simulation Synchronization	get_event	interfaceB.c
Simulation Synchronization	peek_event	interfaceB.c
Simulation Synchronization	simultaneous	interfaceB.c
Simulation Synchronization	max_neq	interfaceB.c
Simulation Synchronization	zapQ_E1	interfaceB.c
Simulation Synchronization	zapQ_E2	interfaceB.c
Simulation Synchronization	pullQ_E1	interfaceB.c
Simulation Synchronization	pullQ_E2	interfaceB.c
Simulation Synchronization	set_time	interfaceB.c
Simulation Synchronization	adv_time	interfaceB.c
Simulation Synchronization	get_time	interfaceB.c
Simulation Synchronization	new_event	interfaceB.c
Simulation Synchronization	show_event	interfaceB.c
Simulation Synchronization	show_event_state	interfaceB.c
Simulation Synchronization	zap_event	interfaceB.c
Simulation Synchronization	setEtime	interfaceB.c
Simulation Synchronization	getEtime	interfaceB.c
Simulation Synchronization	dup_event	interfaceB.c
Simulation Synchronization	pack_event	interfaceB.c
Simulation Synchronization	unpack_event	interfaceB.c

Table 6. Reused Functions for Simulation Synchronization Continued

Simulation Synchronization	setQnum	interfaceB.c
Simulation Synchronization	getQnum	interfaceB.c
Simulation Synchronization	setEid	interfaceB.c
Simulation Synchronization	getEid	interfaceB.c
Simulation Synchronization	setEtype	interfaceB.c
Simulation Synchronization	getEtype	interfaceB.c
Simulation Synchronization	setEent1	interfaceB.c
Simulation Synchronization	getEent1	interfaceB.c
Simulation Synchronization	setEent2	interfaceB.c
Simulation Synchronization	getEent2	interfaceB.c
Simulation Synchronization	setEent3	interfaceB.c
Simulation Synchronization	getEent3	interfaceB.c
Simulation Synchronization	setEpent1	interfaceB.c
Simulation Synchronization	getEpent1	interfaceB.c
Simulation Synchronization	setEpent2	interfaceB.c
Simulation Synchronization	setEpent2	interfaceB.c
Simulation Synchronization	getEpent3	interfaceB.c
Simulation Synchronization	getEpent3	interfaceB.c
Simulation Synchronization	setEdatabuf	interfaceB.c
Simulation Synchronization	getEdatabuf	interfaceB.c
Simulation Synchronization	setEbufsize	interfaceB.c
Simulation Synchronization	getEbufsize	interfaceB.c
Simulation Synchronization	setEdat1	interfaceB.c
Simulation Synchronization	getEdat1	interfaceB.c
Simulation Synchronization	setEdat2	interfaceB.c
Simulation Synchronization	getEdat2	interfaceB.c
Simulation Synchronization	Xinit_safe	interfaceB.c
Simulation Synchronization	Xset_safe	interfaceB.c
Simulation Synchronization	Xadv_safe	interfaceB.c
Simulation Synchronization	Xget_safe	interfaceB.c
Simulation Synchronization	last_time	interfaceB.c
Simulation Synchronization	last_event	interfaceB.c

Table 7. Reused Functions for Simulation Manager

New Module	Function	Battlesim Source
Simulation Manager	init_appl	battle.c
Simulation Manager	do_event	battle.c
Simulation Manager	schedule_init_events	battle.c
Simulation Manager	end_appl	battle.c
Simulation Manager	main	sim_drive.c
Simulation Manager	simdrive	sim_drive.c
Simulation Manager	init_sim_cntrl	sim_cntr.c
Simulation Manager	free_state	sim_cntr.c
Simulation Manager	synch_lps	sim_cntr.c
Simulation Manager	synch_all_lps	sim_cntr.c
Simulation Manager	set_cntrl_cmd	sim_cntr.c
Simulation Manager	get_cntrl_cmd	sim_cntr.c
Simulation Manager	record_LP_state	sim_cntr.c
Simulation Manager	restore_LP_state	sim_cntr.c
Simulation Manager	show_state_list	sim_cntr.c
Simulation Manager	send_update_player	object_mgr.c
Simulation Manager	send_Pcopy_updates	object_mgr.c
Simulation Manager	startup	interfaceB.c

Table 8. Reused Functions for Node/Network Manager

New Module	Function	Battlesim Source
Node/Network Manager	get_LP_node	interfaceB.c
Node/Network Manager	send_event	interfaceB.c
Node/Network Manager	send_both	interfaceB.c
Node/Network Manager	send_object	interfaceB.c
Node/Network Manager	recv_event	interfaceB.c
Node/Network Manager	flush_spectrum	interfaceB.c
Node/Network Manager	true_time	interfaceB.c
Node/Network Manager	setLPid	interfaceB.c
Node/Network Manager	getLPid	interfaceB.c
Node/Network Manager	Send_OS_message	interfaceB.c
Node/Network Manager	Recv_OS_message	interfaceB.c
Node/Network Manager	Wait_for_OS_message	interfaceB.c
Node/Network Manager	Check_for_OS_message	interfaceB.c

Table 9. Reused Functions for I/O Manager

New Module	Function	Battlesim Source
I/O Manager	read_data_line	sim_read.c
I/O Manager	read_data_file	sim_read.c
I/O Manager	init_report	interfaceB.c

Table 10. Reused Functions for Events

New Module	Function	Battlesim Source
Events	predict_event	predict.c
Events	det_boundary_event	predict.c
Events	time_to_intercept_plane	predict.c
Events	time_to_intercept_line	predict.c
Events	time_to_intercept_point	predict.c
Events	det_collision_event	predict.c
Events	det_next_event	schedule.c
Events	new_subclass	methods.c
Events	free_subclass	methods.c
Events	read_subclass	methods.c
Events	copy_subclass	methods.c
Events	list_subclass	methods.c
Events	pack_subclass	methods.c
Events	unpack_subclass	methods.c
Events	subclass_packsize	methods.c
Events	execute_collision	methods.c
Events	det_subclass_internal_event	methods.c
Events	execute_event	ex_event.c
Events	do_dummy	ex_event.c
Events	front_end_object	ex_event.c
Events	center_of_object	ex_event.c
Events	back_end_object	ex_event.c
Events	reached_turnpoint	ex_event.c
Events	collision_distance_reached	ex_event.c
Events	start_player	ex_event.c
Events	update_mapping	ex_event.c
Events	add_player_copy	ex_event.c
Events	update_player_copy	ex_event.c
Events	remove_player_copy	ex_event.c
Events	execute_terminate_object	ex_event.c
Events	do_end	ex_event.c
Events	operator_evaluation	ex_event.c
Events	terminate_object	ex_event.c
Events	on_collision_course	ex_event.c
Events	get_roots	sim_func.c
Events	time_to_intercept	sim_func.c
Events	pull_playercopy_events	sim_func.c
Events	reschedule_playercopy_events	sim_func.c
Events	pull_events_and_reschedule	sim_func.c

Bibliography

1. Banks, Jerry and John S. Carson. *Discrete-Event System Simulation.* Prentice Hall, 1977.
2. Belford, Captain James T. *Object-Oriented Design and Implementation of a Parallel ADA Simulation System.* MS thesis, Air Force Institute of Technology(AU), 1993.
3. Bergman, Captain Kenneth C. *Spatial partitioning of a Battlefield Parallel Discrete-Event Simulation.* MS thesis, Air Force Institute of Technology(AU), 1992. AD-A258911.
4. Chandy, K.M. and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, 440–452 (1979).
5. Committe, DIS Steering, "The DIS Vision A Map to the Future of Distributed Simulation." ftp://sc.ist.ucf.edu/public/STDS/docs/vision.
6. Cormen, Thomas H. et.al. *Introduction to Algorithms.* The MIT Press, 1990.
7. Fujimoto, Richard M. "Parallel and Distributed Simulation." *Proceedings of the 1995 Winter Simulation Conference*, edited by C. Alexopoulos et. al. 118–125. 10662 Los Vaqueros Circle PO Box 3014, Los Altimos CA 90720: IEEE Computer Society Press, December 1995.
8. Garlan, David and Mary Shaw, "An Introduction to Software Architecture." Advances in Software Engineering and Knowledge Engineering, 1993.
9. Hartrum, Tom C., "TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation," 1993.
10. Hiller, Captain James B. *Analytic Performance Models of Parallel Battlefield Simulation Using Conservative Processor Synchronization.* MS thesis, Air Force Institute of Technology(AU), 1994. AD-A289249.
11. Jefferson, D. R. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 404–425 (1985).
12. Masshardt, Captain Conrad P. *Design and Analysis of a parallel Hierarchical Battlefield Simulation.* MS thesis, Air Force Institute of Technology(AU), 1995.
13. Molitoris, Joseph J. and Thomas D. Taylor. "Advanced Simulation, Battle Managers, and Visualization." *Proceedings of the 1995 Winter Simulation Conference*, edited by C Alexopoulos et. al. 10662 Los Vaqueros Circle, PO Box 3014 Los Altimos CA 90720: IEEE Computer Society Press, December 1995.
14. Oswalt, Ivar. "Technology Trends in Miliary Simulation." *Proceedings of the 1995 Winter Simulation Conference*, edited by C. Alexopoulos et. al. 10662 Los Vaqueros Circle, PO Box 3014, Los Altimos CA 90720: IEEE Computer Society Press, December 1995.
15. Painter, Ronald D. "Object-Oriented Military Simulation Development and Application." *Proceedings of the 1995 Winter Simulation Conference*, edited by C Alexopoulos et. al. 10662 Los Vaqueros Circle, PO Box 3014 Los Altimos CA 90720: IEEE Computer Society Press, December 1995.
16. Redden, Joseph J. "Military Simulation and Modeling - Today - Tomarrow." *Proceedings of the 1995 Winter Simulation Conference*, edited by C Alexopoulos et. al. 10662 Los Vaqueros Circle, PO Box 3014 Los Altimos CA 90720: IEEE Computer Society Press, December 1995.

17. Rizza, Captain Robert J. *An Object Oriented Military Simulation Baseline for Parallel Simulation Research.* MS thesis, Air Force Institute of Technology(AU), 1990. AD-A231030.

18. Rumbaugh, James et. al. *Object Oriented Modeling and Design.* Prentice Hall, 1991.

19. Singhal, Mukesh and Niranjan Shivaratri. *Advanced Concepts in Operating Systems.* Mcgraw Hill, 1994.

20. Trachsel, Captain Walter G. *Object Interaction in a parallel in a Parallel Object-Oriented Discrete-Event Simulation.* MS thesis, Air Force Institute of Technolgy(AU), 1993. AD-A274084.

21. Whitted, Gary A., "Software Structural Model Design Methodology for the Modeling Library Components of the Joint Modeling Simulation System (JMASS) Program." ASD/RWWW WPAFB OH.

22. Ziegler, Bernard P., "DEVS Framework for Modeling, Simulation, Analysis and Design of Hybrid Systems." http://linus.cast.uni-linz.ac.at/devs-archive/index.html.

23. Ziegler, Bernard P, "Hierarchical, Modular Discrete-Event Modeling in and Object Oriented Environment." http://linus.cast.uni-linz.ac.at/devs-archive/index.html.

Vita

Captain Kenneth W. Stauffer was born March 24, 1970, in Dayton Ohio. In May 1992 he graduated with a Bachelor of Science in Electrical Enginnering from The Citadel, The Military College of South Carolina and received a commission as a Second Lieutenant in the Air Force.

Captain Stauffer's First assignment was to the Air Force Information Warfare Center at Kelly AFB. as a computer Security Engineer. He served there until May of 1995 when he was accepted to AFIT.

Permanent address: 225 Country Club Dr.
#F1201
Largo, Fl 33771

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Masters Thesis

4. TITLE AND SUBTITLE
An Object-Oriented Discrete-Event Simulation System for Hierarchical Parallel Simulations

5. FUNDING NUMBERS

6. AUTHOR(S)
Kenneth W. Stauffer, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/96D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Mike Lightner
2241 Avionics Circle, Suite 32
WL/AASE BLD 620
Wright-Patterson AFB, OH 45433-7334
(513)255-4429

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The purpose of this research is to design and implement an object-oriented discrete-event simulation system which supports hierarchically constructed players in a parallel or distributed environment. This system design considers modularity and portability so additional modules may be implemented to experiment with new algorithms for both partitoning and synchronization. A simulation system which meets these requirements was partially implemented on an eight-node Intel Hypercube in C. A desired goal was to maintain the functionality of the existing BattleSim application. Test cases used measure the performance and correct operation of the new simulation architecture using a BattleSim subclass. Test results prove correct operation of the new architecture, but show a significant slow down in the parallel operation of this system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel, Simulation, Discrete-Event, Object-Oriented, Hierarchical	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL